https://simplifycpp.org

DRAFT

GCC Internals

System-Level Compilation for Modern C++ on Linux (x86-64)

GNU

Prepared By Ayman Alheraki

GCC Internals and System-Level Compilation for Modern C++ on Linux (x86-64)

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Author's Introduction

The motivation for this work arises from a recurring observation across industry and research: C++ developers routinely rely on compilers as opaque translation engines rather than as deterministic systems with well-defined internal structure and stable behavioral rules. This gap—between the surface practice of writing C++ and the underlying mechanics that give it operational meaning—limits the ability to design efficient data layouts, reason about performance, control binary interfaces, debug complex behavior, or build software that interacts predictably with operating systems and hardware architectures.

My professional work has long existed at the boundary between high-level system architecture and low-level program execution. That boundary is where abstractions meet constraints—where a language's expressive power is tested against real microarchitectural characteristics, memory hierarchies, calling conventions, concurrency models, and the behavior of dynamic loaders and runtime libraries. At this boundary, C++ is not merely a programming language: it is a contract between intent, compiler, processor, operating system, and binary interfaces. Understanding this contract grants precision and confidence; ignoring it introduces fragility and accidental complexity.

GCC plays a critical role in this landscape. It is not only an implementation of C++, but a formal interpreter of C++'s execution semantics. It determines name resolution, lowering rules, exception propagation, object layout, calling convention adherence, symbol visibility, and every performance-relevant transformation applied to code.

GCC translates high-level C++ into the structured intermediate representations that eventually become machine instructions executed on modern x86-64 processors. To understand the behavior of a program, one must understand how GCC reasons, transforms, schedules, and emits that behavior.

This book was written to make that understanding accessible, structured, and technically rigorous. It is not a reference catalogue, nor a survey of compiler engineering history. It is a systematic walk through the compilation pipeline as it applies specifically to Modern C++ in real production environments on Linux x86-64 systems. It connects:

- **Language semantics** to **GIMPLE/SSA form**.
- **Optimization theory** to **register allocation and scheduling**.
- **Binary format mechanics** to **dynamic linking behavior**.
- **Performance models** to **measured pipeline utilization**.
- **ABI rules** to **reliable, long-lived system integration**.

Every transformation in the compiler has a justification, cost model, structural invariant, and measurable runtime effect. Each chapter in this text exposes those relationships directly, with annotated IR, object dumps, pipeline counters, and ABI audits guiding the analysis. The emphasis remains consistent throughout: clarity, correctness, predictability, and verifiable understanding.

This work is intended for experienced software engineers, systems programmers, compiler learners, and architects who build and maintain high-performance or long-lived C++ systems. It does not assume prior compiler implementation experience, but it does assume discipline, patience, and a desire to understand the execution environment deeply rather than heuristically.

C++ remains a language of precise intent. GCC remains one of the most mature and systematically reasoned compilers in active industrial use. Bridging them in a way that is accessible, rigorous, and actionable is the purpose of this book. My hope is that this work enables readers not only to write correct software, but to write software that is *structurally aligned with the machine that will execute it*—software that is stable, performant, maintainable, and engineered with awareness rather than assumption.

— **Ayman Alheraki**

Preface

This book examines the GNU Compiler Collection (GCC) and its role as the formal execution specification for Modern C++ on Linux x86-64 systems. Its central premise is that the compiler is not an implementation detail but the authoritative mechanism that gives operational meaning to source code. The behavior, performance characteristics, binary interfaces, and observable execution of a program emerge from the compiler's translation pipeline, the runtime ABI, and the underlying microarchitectural model. To work effectively at the systems level, a C++ engineer must understand these translation boundaries and the invariants they enforce.

Modern C++ emphasizes explicit semantics, well-defined memory models, and strong guarantees around type behavior and concurrency. These guarantees are only realized through a coordinated stack: source-level constructs lowered into structured intermediate representation (GIMPLE/SSA), optimized through target-aware transformations, expressed in RTL, and finally materialized as executable machine code. GCC implements this pipeline with precise invariants about calling conventions, exception handling, object layout, alignment, and symbol visibility. These invariants constitute a contract that defines binary compatibility and execution correctness across kernels, shared libraries, CPUs, and compiler versions.

This text approaches GCC as a multi-layered system:

- **The Frontend** resolves C++ semantics—name lookup, instantiation, overload

resolution, concepts, constant evaluation—and constructs a canonical semantic graph.

- **The Midend** applies machine-independent transformations over GIMPLE and SSA, performing scalar optimization, alias analysis, value propagation, loop transformation, vectorization, and inlining under a cost model.
- **The Backend** lowers optimized IR to RTL and schedules instructions according to microarchitectural constraints, register availability, execution port topology, and memory hierarchy behavior.
- **The Linker and Loader** finalize symbol bindings, resolve dynamic dependencies, assign memory protections, and construct the execution address space.
- **The Runtime and ABI** define object representation, exception unwinding, RTTI structures, TLS models, and library integration rules that preserve system interoperability.

This structure is presented not as static documentation, but as a basis for disciplined engineering. Every chapter advances from a formal rule to its observable impact in generated code, and from generated code to its performance implications on real hardware. Wherever appropriate, examples include:

- GIMPLE and RTL dumps to trace compiler decisions,
- Disassembly annotated with pipeline scheduling considerations,
- perf-based execution metrics to validate optimization outcomes,
- ABI verification to ensure long-term binary stability.

The goal is not to modify GCC into a different compiler, but to develop the fluency required to work alongside it: to anticipate transformations, to diagnose suboptimal code generation, to design data layouts that align with vector execution, to produce stable binary interfaces, and to reason about performance directly from the compiler's intermediate forms.

This book assumes the reader is comfortable with Modern C++ syntax, pointer model semantics, and Unix systems programming. No previous compiler implementation experience is required, but familiarity with ELF, paging, and x86-64 machine code is beneficial. The presentation is precise and complete, but not abstracted away from practice; each concept is tied to concrete output and measurable system behavior.

The objective is to equip advanced C++ practitioners, system software engineers, and compiler-adjacent researchers with a practical and exact understanding of how GCC translates intent into execution. With this understanding, one can write C++ that is not only correct and portable, but structurally aligned with the machine that will run it.

Part I

THE GNU COMPILATION MODEL AND SYSTEM CONTRACTS

Chapter 1

The Compiler as the System's Formal Execution Specification

1.1 The Compiler Defines Semantics, Not the Source Language

The C++ language specification describes program behavior in terms of an *abstract machine*. However, execution does not occur in this abstract model; it occurs on a specific hardware and operating system architecture. The responsibility of *defining the executable meaning* of a C++ program therefore belongs not to the language text itself, but to the **compiler implementation**. Under modern Linux on x86-64, GCC is the component that translates the abstract semantics of C++ into a concrete, verifiable, and executable form. The compiler is the mechanism that determines how language constructs map to memory, control flow, calling conventions, binary interfaces, and optimization constraints.

In effect, **the compiler is the semantic authority**. The programmer writes source

code, but the compiler determines what that program *is* when executed.

1.1.1 Source Code is Not Executable Specification

C++ source code contains high-level descriptions of computation, but it omits:

- Memory layout decisions
- Register allocation and operand movement rules
- Elision, merging, and reordering of computations
- Link-time symbol visibility constraints
- Instruction scheduling and microarchitectural placement

These omissions are intentional. The language standard relies on the compiler to resolve these details. As a result, the *run-time behavior* of a program is defined not by the written code, but by the compiler's interpretation and transformation of that code.

1.1.2 Semantic Lowering and Transformation Phases

GCC establishes program meaning through a multi-phase reduction pipeline:

1. **Parsing and Semantic Analysis**
   Template instantiation, overload resolution, constant evaluation, and type deduction fix the high-level structure.

2. **GIMPLE SSA Transformation**
   The program is reduced to a structured, side-effect-constrained representation. This is the level at which most semantic-preserving optimizations occur.

3. **RTL and Instruction Selection**
   Language semantics are translated into machine-specific effects: register-class choices, memory addressing modes, and control-transfer forms.

4. **Emission and Relocation**
   Symbolic references and binary interfaces are committed to the ELF object representation. This finalizes observable calling conventions and linkage behavior.

At each lowering stage, **semantic meaning is refined**, not merely translated.

1.1.3 The Role of Undefined and Implementation-Defined Behavior

C++ intentionally introduces cases where program behavior is not fully defined by the language specification. GCC resolves these cases by mapping them to deterministic machine-level effects, subject to optimization. For example:

- Pointer aliasing assumptions influence load/store reordering.
- Signed integer overflow is treated as undefined, enabling algebraic simplifications.
- Object lifetime boundaries determine whether constructors and destructors can be eliminated.

The executable program therefore reflects **compiler-governed semantics**, not a naïve reading of source text.

1.1.4 The Compiler as the Formal Boundary of Program Reality

The compiled binary expresses:

- The exact memory and register behavior of each instruction.

- The enforced calling convention across function boundaries.
- The structure of unwind tables, exception propagation, and stack discipline.
- Symbol linkage visibility and relocation behavior.

These properties define what the program *is* at execution.
No interpretation of the original source can override this definition.

1.1.5 Consequence for System-Level C++ Engineering

For performance-critical or correctness-critical systems, reasoning must occur at the level of **compiler-controlled behavior**, not source representation. This requires:

- Inspection of GIMPLE and RTL to understand semantic reduction.
- Awareness of ABI-level constraints rather than informal calling assumptions.
- Analysis of generated machine code for actual execution behavior.
- Use of compiler flags and attributes to regulate optimization domains.

A program's meaning is therefore defined by **the compiler's lowering and optimization decisions**, which embody the true operational semantics of the architecture.

1.2 The Toolchain as the System's Deterministic Behavioral Model

In a Linux x86-64 environment, a compiled C++ program does not execute in isolation. It executes within a controlled and layered system composed of the compiler, assembler, linker, dynamic loader, and runtime libraries. Together, these components define the **deterministic behavioral model** under which the program operates. The toolchain is therefore not a build utility; it is the mechanism by which program semantics are instantiated, validated, and constrained at execution time.
The system's behavior is determined by three forms of specification:

1. **Language-level rules** (C++ abstract machine and memory model)
2. **ABI and binary interface constraints** (System V AMD64 ABI, Itanium C++ ABI)
3. **Toolchain lowering and runtime enforcement** (GCC, binutils, glibc, ld.so)

The interaction of these layers defines the program's real behavior.
Any correct analysis of execution must be performed across this full system, not solely at the source level.

1.2.1 Determinism Through Standardized Execution Contracts

The compiler enforces deterministic program behavior through standardized contracts that remain unchanged across compilation units and library boundaries:

- **Calling conventions** define how parameters and return values are passed.
- **Exception ABI** defines how unwinding and stack frame recovery operate.

- **Object layout rules** define memory representation of class and polymorphic types.
- **Linkage visibility rules** determine whether symbols may be substituted or inlined.

These rules ensure that compiled components remain interoperable regardless of optimization level, compilation order, or target microarchitecture.

1.2.2 Determinism at the Code Generation and Linking Boundary

Once the compiler lowers semantically reduced IR to machine-oriented RTL, observable behavior becomes fixed in terms of:

- Instruction selection
- Register assignment
- Control-flow and branch layout
- Memory access patterns

The linker then determines the global symbol resolution topology and address space layout.
As of post-2020 ELF and glibc evolution, this layout is fully reproducible when:

```
Compiler version + Flags + Input + Link order = constant
```

This is a **strict determinism guarantee**, enabling binary-level reproducibility and distributed system deployment stability.

1.2.3 Runtime Enforcement of Toolchain Semantics

The dynamic loader (`ld.so`) enforces correct resolution of shared objects, relocation entries, TLS regions, and PLT/GOT binding behavior. glibc enforces runtime invariants such as:

- Thread-local storage consistency
- C++ constructor and destructor sequencing via `.init_array` and `.fini_array`
- Exception propagation compliance with unwind tables
- Standard memory model ordering semantics

The toolchain therefore defines **how** execution proceeds, not simply how code is built.

1.2.4 Implications for System-Level C++ Engineering

Because the toolchain defines the operational semantics of execution:

- Performance analysis must include compiler output inspection.
- Concurrency correctness must be validated against ABI and memory model guarantees.
- Binary compatibility must be reasoned through symbol visibility and linkage constraints.
- Real execution is defined at the granularity of the compiled binary, not the source.

A system written in C++ is not defined by its source texts alone.
It is defined by the constraints and behaviors imposed by the **compiler + linker + loader + runtime** composite system.

1.2.5 Summary

The GNU toolchain does not merely transform code; it **defines the executable meaning** of C++ programs on Linux x86-64. It establishes deterministic execution rules, constrains permissible program behavior, and ensures binary compatibility across software boundaries and hardware generations. The observable semantics of a system are therefore properties of the **compiled artifact and toolchain model**, not solely of the source language.

1.3 Visibility, Inspectability, and Reproducibility as Engineering Requirements

For a compiled C++ program to be correct in a system-level environment, its behavior must be **observable, analyzable, and repeatable** across builds and deployments. Visibility into the compilation pipeline, inspectability of intermediate representations and emitted binaries, and reproducibility of build outputs are not convenience features; they are **engineering requirements** that ensure program correctness, diagnosability, and long-term maintainability in GNU/Linux environments.

1.3.1 Visibility into the Compilation Pipeline

The GNU toolchain is designed to expose each stage of program lowering:

- GIMPLE and SSA dumps show structural program representation.
- RTL dumps show machine-oriented transformation states.
- Assembly listings show the final scheduled instruction stream.

This transparency is required because semantic meaning is **established by the compiler**, not by source code. Inspection allows engineers to verify:

- Whether optimizations preserved intended ordering constraints.
- Whether aliasing assumptions match real data access patterns.
- Whether control-flow transformations preserved required invariants.

Visibility ensures the ability to reason about the **actual executable artifact**, not a conceptual or intended behavior.

1.3.2 Inspectability at the Binary Interface Level

Once compiled, program behavior is mediated by:

- ELF section layout
- Relocation records
- Symbol visibility attributes
- PLT/GOT dispatch paths
- Itanium C++ ABI type and vtable representations

Inspectability ensures that:

- External linkage boundaries remain stable.
- ABI correctness can be validated mechanically.
- Performance-critical indirect calls can be traced and analyzed.
- Exception and unwinding state structures can be reconstructed reliably.

Without inspectable binary artifacts, debugging and performance analysis become non-deterministic.

1.3.3 Reproducibility as a Deterministic Execution Property

Reproducibility ensures that identical source, toolchain, flags, and inputs produce an identical binary bitstream. This requirement is central to:

- Distributed deployment across heterogeneous environments.

- Rollback-capable update mechanisms.
- Verification of supply chain integrity.
- Performance regression analysis based on binary identity.

Post-2020 GCC, binutils, and glibc revisions incorporate measures to ensure reproducibility by removing nondeterministic sources such as timestamp metadata and unordered symbol table emission.
Reproducibility is not static artifact equivalence alone; it is the guarantee of **semantic identity across rebuilds**.

1.3.4 Stability Under Optimization and Microarchitectural Change

The toolchain must maintain stable observable semantics even when:

- Optimization heuristics evolve between compiler releases.
- Instruction scheduling changes due to updated cost models.
- Target generation changes (e.g., Skylake → Zen → Sapphire Rapids).

The boundary of stability is the **ABI**, not the generated instruction sequence.
Thus, reproducibility applies to:

- Control transfer behavior visible at symbol boundaries.
- Object layout and calling conventions.
- Exception and TLS state models.

Performance characteristics may vary; **behavior may not**.

1.3.5 Engineering Outcome

Visibility, inspectability, and reproducibility ensure that:

- Program semantics remain explainable.
- Execution behavior remains verifiable.
- Optimization decisions remain analyzable.
- Deployment behavior remains stable across time and hardware.

These properties allow system-level C++ software to be **reasoned about**, not merely compiled and executed.

1.4 Stability Contracts Across CPU Generations and OS Versions

A compiled C++ program must remain **binary-correct and execution-consistent** across hardware revisions and operating system updates. These guarantees are not provided by the source language but by the **ABI contracts** and the system's runtime execution environment. The GNU toolchain enforces these stability constraints so that the meaning of a compiled program remains invariant even as compilers evolve and microarchitectures differ.
The stability boundary is defined where **compiler-controlled semantics intersect with externally observable behavior**: the ABI, the ELF format, exception frames, calling conventions, and runtime initialization sequences.

1.4.1 ABI as a Fixed External Contract

The System V AMD64 ABI and the Itanium C++ ABI define:

- Function calling conventions and register assignments.
- Parameter passing and return value placement.
- Stack layout and alignment constraints.
- Object representation, including vtable layouts and RTTI structures.
- Exception propagation and unwinding data formats.

These rules ensure that:

- Binaries produced by different compiler versions interoperate.

- Shared libraries and dynamically linked executables remain callable.
- The same object layout is used across CPU generations.

The ABI is therefore the **non-negotiable stability surface** of compiled C++ on Linux.

1.4.2 Microarchitectural Variation Without Semantic Change

Modern x86-64 CPUs differ in:

- Execution pipelines and reorder buffer width.
- Branch prediction and speculative execution heuristics.
- Instruction throughput and latency characteristics.
- Cache hierarchy and memory-level parallelism properties.

However, these differences cannot affect:

- The values stored in memory at defined sequence points.
- The calling convention-visible register state at function boundaries.
- The preservation of callee-saved registers.
- The binary identity of exported symbols and type layouts.

The compiler may generate specialized instruction sequences using `-march` and `-mtune`, but **semantic correctness remains invariant across all permitted target microarchitectures** that conform to the ABI.

1.4.3 Runtime Library and Kernel Interface Continuity

glibc and the Linux kernel present stable runtime interfaces that ensure:

- System call interface correctness across kernel releases.
- Thread-local storage layout consistency.
- Reliable exception propagation across shared library boundaries.
- Compatibility of dynamically loaded modules compiled at earlier toolchain revisions.

Post-2020 glibc consolidations (notably the integration of system call stubs and elimination of `libpthread` as a separately linked component) preserve ABI surfaces while restructuring internal runtime paths for improved determinism and reproducibility.
The stability contract is therefore **semantic**, not internal implementation–based.

1.4.4 Compiler Evolution Under Stability Constraints

GCC continues to evolve in:

- Optimization heuristics.
- Vectorization and auto-parallelization support.
- Alias and dependency analysis precision.
- Inlining cost modeling and profile-guided code layout.

However, these transformations cannot:

- Alter externally visible binary formats.
- Change calling conventions.
- Modify object layouts or vtable structures.
- Break compatibility with previously linked binaries.

The compiler is free to improve the **internal execution efficiency**, but not the **externally observable semantics**.

1.4.5 Practical Engineering Implication

For long-lived C++ systems:

- Performance portability requires architecture-tuning flags, not code changes.
- Binary compatibility is ensured through adherence to ABI-visible constructs.
- Forward execution safety relies on stable runtime and loader interfaces.
- Verification and debugging require reasoning at the ABI and ELF boundary.

Therefore, **stability across CPU and OS generations is not incidental; it is engineered through fixed interface contracts** that govern compilation, linking, loading, and runtime execution.

1.5 Examples: ABI Continuity Analysis Across GCC Major Versions

Application Binary Interface (ABI) continuity ensures that independently compiled components remain interoperable across compiler versions. The GNU toolchain maintains strict ABI stability on Linux x86-64 for externally visible constructs governed by the System V AMD64 ABI and the Itanium C++ ABI. This section provides structured examples demonstrating that binaries compiled with different GCC major versions link and execute correctly when adhering to ABI-visible constructs.
The examples use post-2020 GCC releases (e.g., GCC 10, 12, 13, and 14 series) and glibc 2.34 or later.

1.5.1 Stable Class Layout and Virtual Dispatch Across Versions

Library — compiled with GCC 10:

```
// lib.cpp
struct S {
    int x;
    virtual int f() const { return x + 1; }
};

extern "C" int dispatch(const S* p) {
    return p->f();
}
```

Compile:

```
g++-10 -O2 -fPIC -shared lib.cpp -o libabi.so
```

Executable — compiled with GCC 14:

```cpp
// main.cpp
#include <iostream>

struct S {
    int x;
    virtual int f() const { return x + 1; }
};

extern "C" int dispatch(const S*);

int main() {
    S obj{29};
    std::cout << dispatch(&obj) << "\n";
}
```

Compile and link:

```
g++-14 -O2 main.cpp libabi.so -o test
```

Expected Runtime Output:

```
30
```

Analysis:

- `sizeof(S)`, vtable layout, and member offsets remain identical across GCC 10–14.
- The call to `f()` is performed through the Itanium ABI vtable dispatch contract.
- No recompilation of the shared library is required.

This confirms **ABI-stable polymorphic type layout**.

1.5.2 Name Mangling and Symbol Binding Stability

For non-templated functions with external linkage:

```
// stable.cpp
namespace A { namespace B {
    int add(int a, int b) { return a + b; }
} }
```

Compile with multiple GCC versions:

```
g++-10 -c stable.cpp -o stable10.o
g++-14 -c stable.cpp -o stable14.o
```

Inspect symbols:

```
nm -C stable10.o | grep add
nm -C stable14.o | grep add
```

Expected symbol in both:

```
int A::B::add(int, int)
```

Mangling (Itanium ABI):

```
_ZN1A1B3addEii
```

Name mangling remains stable because it is specified externally by the Itanium ABI, not by GCC.

1.5.3 Exception Propagation Compatibility Across Toolchain Versions

Throwing Library (compiled with GCC 11):

```
// thrower.cpp
#include <stdexcept>

extern "C" void trigger() {
    throw std::runtime_error("failure");
}
g++-11 -O2 -fPIC -shared thrower.cpp -o libthrow.so
```

Catching Executable (compiled with GCC 14):

```
// catcher.cpp
#include <iostream>
#include <stdexcept>

extern "C" void trigger();

int main() {
    try {
        trigger();
    } catch(const std::exception& e) {
        std::cout << "caught: " << e.what() << "\n";
    }
}
g++-14 catcher.cpp libthrow.so -o test
```

Expected Runtime Output:

```
caught: failure
```

Reason:

- Exception objects use a stable runtime type information (RTTI) encoding.
- Stack unwinding format is defined by DWARF `.eh_frame` and Itanium EH ABI.
- The compiler does not change exception layout across major versions.

This validates **exception ABI continuity**.

1.5.4 Function Call Boundary Invariance Under Optimization Evolution

Examine call-site assembly differences while preserving boundary semantics:

```
g++-10 -O2 foo.cpp -S -o foo10.s
g++-14 -O2 foo.cpp -S -o foo14.s
```

Expected:

- Instruction sequences differ due to improved scheduling and vectorization.
- **Call boundaries and register passing remain unchanged**, reflecting ABI stability.

1.5.5 Summary of Findings

Feature	Stability Mechanism	Confirmed Across Versions
Class layout & vtable structure	Itanium C++ ABI	Yes
Name mangling	ABI-mandated encodings	Yes
Function calling convention	System V AMD64 ABI	Yes
Exception propagation behavior	DWARF EH + Itanium EH ABI	Yes
Compiler optimization strategies	Vary across versions	Does not affect ABI

ABI surfaces are stable; optimization behavior is permitted to evolve.

Chapter 2

The Linux Execution Stack and Boundary Interfaces

2.1 CPU → Kernel → Loader → Runtime → Application Execution Path

Execution of a compiled C++ binary on Linux x86-64 proceeds through a structured transition across processor privilege levels, memory initialization phases, and runtime activation layers. The compiler and toolchain assume that these transitions occur in a well-defined order. Therefore, understanding the execution path is necessary for analyzing correctness, performance behavior, initialization ordering, and ABI-level stability.

Execution proceeds through five deterministic stages:

```
(1) CPU Reset / Privileged Entry
(2) Kernel Process Image Construction
(3) Dynamic Loader Activation (ld.so)
```

```
(4) C and C++ Runtime Initialization
(5) Transfer of Control to main()
```

Each stage defines constraints that the compiler relies upon when lowering high-level C++ semantics into machine code.

2.1.1 CPU Architectural Preconditions

When user-space execution begins, the CPU operates in **long mode** (64-bit), with the **System V AMD64 ABI** register contract already in effect. At `_start`, the following properties are guaranteed:

- `%rsp` is initialized to the top of the user stack.
- Function parameter registers (`%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`) have defined calling semantics.
- Caller-saved and callee-saved register classes are stable.
- Memory protection and paging are enabled.

Compiler-generated code depends on these invariants. No C++ runtime logic is active yet.

2.1.2 Kernel: Process and Address Space Construction

When a process is launched through `execve()`, the Linux kernel performs:

1. Creation of a new virtual memory layout.
2. Loading of segments defined in the program's ELF program headers.

3. Initialization of stack contents with `argc`, `argv`, `envp`, and the auxiliary vector (`auxv`).
4. Selection of the **dynamic loader entry point** as the initial instruction pointer for dynamically linked binaries.

Notably:

- No C++ constructors run at this stage.
- No dynamic relocations are yet applied.
- No user code executes directly from ELF entry unless the executable is statically linked.

This stage defines the memory environment that the compiler assumes for correct program layout and relocation resolution.

2.1.3 Loader: Dynamic Linking and Relocation (ld.so)

The dynamic loader (`ld.so`) is the first userspace instruction stream executed. It performs:

- Mapping of required shared libraries.
- Application of relocations recorded in `.rela.dyn` and `.rela.plt`.
- Construction of the Global Offset Table (GOT) and Procedure Linkage Table (PLT).
- Initialization of Thread-Local Storage (TLS) descriptors.
- Parsing of auxiliary vector entries for CPU feature dispatch and VDSO enablement.

Position-Independent Executable (PIE) model, standard in post-2020 distributions, ensures that the loader relocates both executables and libraries in a unified manner. At completion, the loader transfers control to C runtime initialization.

2.1.4 Runtime: libgcc + glibc + C++ Initialization

The entry sequence typically follows:

```
_start → __libc_start_main → __libc_start_call_main → main
```

Before `main()` is invoked:

- The C standard library initializes locale, threading, and I/O subsystems.
- The C++ runtime executes `.init_array` to construct static storage-duration objects.
- Destructor callbacks are registered through `__cxa_atexit`.
- Exception-unwinding metadata (`.eh_frame`) is registered with the runtime system.

This phase establishes all **invariants assumed by C++ language semantics**, including object lifetime, exception propagation guarantees, and TLS consistency.

2.1.5 Application Execution Under Compiler-Defined Semantics

Only after the runtime completes initialization does control transfer to:

```
int main(int argc, char** argv)
```

At this point:

- ABI contracts are fully active.
- All dynamic symbol bindings are resolved (unless lazy PLT binding remains configured).
- All static objects are in valid constructed state.
- Memory allocation, exception semantics, and thread behavior adhere to glibc and libgcc runtime models.

Execution from this point onward is governed entirely by the **machine code emitted by the compiler**, constrained by the ABI, and executed under the protection and scheduling mechanisms of the kernel.

2.1.6 Summary

The execution path enforcing correct program behavior is:

Stage	Responsibility	Guarantees Relevant to C++ Execution
CPU	Architectural state and calling convention	Register ABI and execution mode correctness
Kernel	Memory image creation and stack initialization	Deterministic ELF segment mapping
Loader (ld.so)	Relocation and dynamic symbol binding	GOT/PLT correctness and TLS setup
glibc + libgcc	Runtime and static object initialization	Proper object lifetime, exception, and memory model invariants

Stage	Responsibility	Guarantees Relevant to C++ Execution
Application	Program logic execution	Compiler-defined semantics

Correct reasoning about compiled C++ behavior requires analyzing execution **across these layers**, not solely at the level of source code or generated assembly.

2.2 System Call ABI Calling Convention and Register Assignments

The transition from user-space to kernel-space execution is not performed using the standard function calling convention defined for C++ user-space code. Instead, Linux on x86-64 defines a **distinct System Call ABI**, optimized for privilege transition and kernel entry handling. GCC does not directly emit system call instructions in high-level C++ code; instead, glibc wrapper functions marshal parameters according to the System Call ABI. However, understanding the ABI is essential for analyzing system-level behavior, performance, and register state during debugging.
This ABI is **mandatory**. Any deviation results in undefined behavior at the processor–kernel boundary.

2.2.1 Register Assignment for System Calls

The System Call ABI defines the following calling convention:

Purpose	**Register**
System call number	%rax
Argument 1	%rdi
Argument 2	%rsi
Argument 3	%rdx
Argument 4	%r10
Argument 5	%r8
Argument 6	%r9

Purpose	Register
Return value	%rax

Notably:

- `%rcx` and `%r11` are **always clobbered** by the `syscall` instruction.
- All other registers follow standard System V AMD64 ABI preservation rules across the boundary.
- The stack pointer (`%rsp`) must be valid and 16-byte aligned at the call site.

This convention is chosen to avoid register conflict with the kernel's internal state during ring transition.

2.2.2 The `syscall` Instruction and Privilege Transition Sequence

The `syscall` instruction performs:

1. **Privilege level change** from ring 3 to ring 0.
2. **Instruction pointer override** using the model-specific register `IA32_LSTAR` (kernel entry point).
3. **Masking of flags** using `IA32_FMASK`.
4. **Storage of return address and RFLAGS** in `%rcx` and `%r11`.

This transition enforces **serialization of speculative execution state**, which introduces non-trivial latency characteristics on post-Skylake microarchitectures. The compiler must therefore treat system call boundaries as **ordering fences**. No reordering of memory operations across the boundary is allowed.

2.2.3 System Call ABI vs. User-Space ABI

The System V AMD64 ABI used in C++ function calls differs from the System Call ABI. Key distinctions:

Feature	User-Space ABI	System Call ABI
Parameter registers 1–4	%rdi, %rsi, %rdx, %rcx	%rdi, %rsi, %rdx, %r10
%rcx usage	Parameter and call-clobber register	Reserved for return RIP storage (clobbered)
Calling instruction	`call` and `ret`	`syscall` + kernel-managed return
Stack frame state	Managed by caller/callee	Kernel constructs separate privileged stack

Because `%rcx` is clobbered during a `syscall`, **glibc must move user-space `%rcx` into `%r10`** before invocation. Compilers rely on glibc to generate these move sequences correctly.

2.2.4 Consequences for Compiler Lowering and Optimization

The compiler treats system calls as **opaque side-effecting operations**:

- The call cannot be inlined or memoized.
- No algebraic or control-flow simplification may cross the boundary.
- Memory ordering is implicitly sequentially consistent with respect to the kernel.
- Register allocation must preserve the ABI assignments exactly.

This makes system call sites **optimization barriers**.
From a code-generation standpoint:

```
mov $60, %rax        ; SYS_exit
mov $0, %rdi
syscall
```

is not equivalent to:

```
return;
```

The former creates a mode transition; the latter terminates program execution according to C++ semantics.

2.2.5 Engineering Implications

For system-level C++:

- **Latency minimization** requires reducing system call frequency.
- **Concurrency correctness** requires awareness of the enforced memory ordering boundary.
- **Debugging correctness** requires distinguishing user register state from kernel-override registers.
- **Performance profiling** must treat system call boundaries as pipeline flush events.

Thus, the System Call ABI defines the **precise execution boundary** between compiler-governed semantics and kernel-governed semantics.

2.3 The `syscall` Instruction and VDSO Acceleration Layer

The `syscall` instruction defines the hardware-supported mechanism for transitioning from user mode to kernel mode. The transition enforces a privilege change, architectural state preservation, and register masking. While necessary for accessing kernel services, `syscall` is **latency-expensive** due to pipeline serialization and supervisor-mode setup. To mitigate these costs for frequently invoked kernel-adjacent operations, Linux exposes a **Virtual Dynamic Shared Object (VDSO)** region that allows certain kernel-maintained computations to be executed **entirely in user space**, eliminating the privilege transition overhead.

2.3.1 Execution Semantics of the `syscall` Instruction

When executed, `syscall` performs:

1. Privilege level switch: ring 3 $\rightarrow$ ring 0.
2. Instruction pointer load from `IA32_LSTAR`.
3. `RFLAGS` masking through `IA32_FMASK`.
4. Saving of user return address and flags into `%rcx` and `%r11`.

Its effect is equivalent to a **synchronous, ordered control transfer** into the kernel entry point. As a side effect:

- The speculation pipeline is partially or fully serialized.
- `%rcx` and `%r11` are always overwritten.

- TLB, BTB, and RSB mitigation rules may introduce additional synchronization cost.

Thus, the compiler and optimizer must treat `syscall` as a **full ordering and execution boundary**.

2.3.2 Performance Characteristics on Post-2020 x86-64 CPUs

On recent server-class and workstation-class microarchitectures (e.g., Skylake-X, Zen 3, Sapphire Rapids), `syscall` latency typically falls in the **150–350 cycle** range. Variation depends on:

- Speculation barrier configuration.
- Kernel-level hardening (SMEP, IBRS, STIBP).
- TLB locality for kernel trampolines and thread stacks.

This cost is significant relative to typical user-level function call latency (3–12 cycles). Thus, system call frequency and granularity become critical design parameters for performance-sensitive C++ applications.

2.3.3 VDSO: User-Space Execution of Kernel-Managed Functions

The **VDSO (Virtual Dynamic Shared Object)** is a memory-mapped region populated by the kernel at process start. It contains **non-privileged implementations** of specific timekeeping and CPU-topology-related functions, backed by kernel-maintained state in shared memory.
Examples routinely dispatched through VDSO include:

- `clock_gettime()`
- `gettimeofday()`
- `time()`
- `getcpu()`

The VDSO implementation:

- Executes in user mode (no privilege transition).
- Reads kernel-updated data structures ensured coherent via cache synchronization mechanisms.
- Uses load-acquire semantics and sequence counter validation to ensure reliability.

Typical call latency becomes **tens of cycles**, not hundreds.

2.3.4 Loader and Runtime Binding Behavior

The dynamic loader resolves VDSO symbols at runtime. The glibc implementation of system call wrappers:

1. Attempts to resolve a corresponding `__vdso_*` symbol.
2. If available, dispatches directly into the VDSO function.
3. If unavailable (e.g., nonstandard kernel, restricted environment), falls back to the `syscall` ABI.

The compiler is **not aware** of whether a given call will use VDSO or `syscall`; dispatch selection occurs strictly at runtime via relocation binding.
Thus, VDSO does not modify compiler behavior—it modifies **where** the generated call targets execute.

2.3.5 Implications for System-Level C++ Execution

Aspect	`syscall` Path	VDSO Path
Privilege mode switch	Yes	No
Register clobber semantics	Enforced by kernel	User-space call ABI only
Latency	High (pipeline flush)	Low (local memory access)
Determinism	Fully deterministic	Deterministic under page mapping invariants
Compiler involvement	None beyond ABI lowering	None; resolution occurs at runtime

For performance-critical C++ systems:

- Prefer operations offered through VDSO where available.
- Avoid manually emitting `syscall` unless implementing low-level OS interfaces.
- Treat system call execution boundaries as performance and scheduling inflection points.

The `syscall` and VDSO layers together define **the operational interface limit between compiler-governed execution semantics and kernel-governed execution semantics**.

2.4 Userspace Loader (ld.so) as a Policy Engine

The dynamic loader (`ld.so`) is not merely a relocation executor; it is a **policy engine** that establishes the operational environment under which the compiled binary executes. It determines dynamic symbol binding behavior, relocation strategy, address space layout, thread-local storage configuration, and library dependency ordering. Its decisions directly influence performance, correctness, and the interpretation of linking constraints encoded during compilation.
The loader operates at the boundary where **static compilation artifacts** become **executing software**. Therefore, its behavior defines part of the executable program semantics.

2.4.1 Loader Responsibilities as Defined by ELF Semantics

Upon process start, `ld.so` is entered prior to any C++ runtime initialization. It performs:

1. **Mapping shared library dependencies** via program header information and DT_NEEDED entries.
2. **Constructing the dynamic link map**, establishing the load and lookup order of shared objects.
3. **Resolving relocations**, including `.rela.dyn` and `.rela.plt` entries.
4. **Constructing the Global Offset Table (GOT)** and populating PLT stubs.
5. **Configuring Thread-Local Storage (TLS)** layout and per-thread initial state.
6. **Activating VDSO routines** by interpreting `AT_SYSINFO_EHDR` auxiliary vector data.

These operations finalize symbol visibility and memory topology assumptions made by GCC and the linker.

2.4.2 The Loader as the Enforcement Point for Symbol Resolution Policy

Symbol lookup obeys a **strict deterministic order**:

```
Executable → Dependencies in link order → LD_PRELOAD entries → Global namespace
```

This ordering ensures:

- ABI-stable linking behavior across library versions.
- Predictable override semantics for interposable symbols.
- Avoidance of symbol collisions through deterministic resolution.

Loader ordering rules directly constrain what optimizations the compiler may apply. If a function may be interposed at runtime, GCC must generate a **PLT-based indirect call**, prohibiting inlining and constant propagation across the boundary. This makes loader behavior an **optimizer boundary**.

2.4.3 Loader as the Authority for PIE and ASLR Execution Layout

Post-2020 mainstream Linux distributions build executables as **PIE** by default. Under PIE:

- Code segments have no fixed absolute address.
- All symbol references use RIP-relative addressing.

- The loader selects the actual load address at runtime.

This enables **ASLR (Address Space Layout Randomization)**, but requires:

- The loader to apply full relocation on load.
- The compiler to emit position-independent code sequences.

Thus, the loader defines the **final instruction pointer mapping** under which the compiled code executes.

2.4.4 TLS Model Selection and Enforcement

The loader allocates and initializes **Thread-Local Storage** regions based on the TLS model selected at compile time (`local-exec`, `initial-exec`, `local-dynamic`, `global-dynamic`). This has direct consequences on:

- Code generation for `thread_local` variables.
- Access method lowering (RIP-relative or TLS descriptor based).
- Performance characteristics of thread-local lookups.

The compiler assumes the loader's TLS configuration is correct; if mismatched, behavior fails at runtime, not compile time.

2.4.5 Loader as the Gatekeeper for Runtime Feature Dispatch

The loader interprets hardware capability descriptors (`AT_HWCAP`, `AT_HWCAP2`) and configures glibc and auxiliary runtime routines to select optimized execution paths (e.g., vectorized `memcpy`, lock-free atomics fast-paths). These dispatches occur **before** user code executes.
Thus, final performance characteristics are determined by **loader-selected runtime implementations**, not solely by compiler optimization.

2.4.6 Summary

Responsibility	Loader Role	Compiler Dependency
Symbol binding	Enforces global lookup and interposition	Governs inlining and call lowering
Relocation	Finalizes address values and GOT/PLT targets	Validates position-independent code assumptions
TLS initialization	Establishes per-thread storage model	Enables correct `thread_local` access semantics
Execution environment selection	Applies VDSO and CPU feature dispatch	Controls optimized runtime routines
Memory layout	Assigns load addresses under ASLR/PIE	Guarantees RIP-relative relocation correctness

`ld.so` is therefore a **semantic enforcement layer**, not a loader in a minimal sense. It completes the compilation pipeline by translating link-time assumptions into executable program invariants.

2.5 Examples: Disassembling `_start` → `__libc_start_call_main`

To understand how execution transitions from the ELF entry point to user code, we analyze the disassembly and call sequence between the program's `_start` symbol, the runtime entry routine `__libc_start_main`, and its internal wrapper `__libc_start_call_main`. This path is present in all dynamically linked C++ executables produced by GCC on Linux x86-64, regardless of optimization level or language features.
The example program is minimal to remove irrelevant noise:

```
// example.cpp
int main() { return 0; }
```

Compile with symbols and no frame omission:

```
g++ -O2 -fno-omit-frame-pointer -g example.cpp -o example
```

2.5.1 `_start`: Entry Point Defined by the Linker

Disassemble the entry:

```
objdump -d example | grep -A20 "<_start>"
```

Representative output (glibc 2.34+):

```
_start:
    xor    %rbp,%rbp
    mov    %rdx,%r9          ; r9 = auxiliary vector (envp tail boundary)
    pop    %rsi              ; rsi = argv
    mov    %rsp,%rdx         ; rdx = envp pointer (after argv)
```

```
    and     $~0xf,%rsp          ; enforce stack alignment
    call    __libc_start_main@PLT
```

Key properties:

- `%rsp` is aligned to **16 bytes**, as required by the ABI.
- Registers are arranged to conform to the calling convention expected by `__libc_start_main`:
  - `%rdi` implicitly holds `main`.
  - `%rsi` holds `argv`.
  - `%rdx` holds `envp`.

This routine performs **no runtime initialization**. It only forwards control.

2.5.2 `__libc_start_main`: Runtime Coordinator

Locate definition:

```
objdump -d /lib/x86_64-linux-gnu/libc.so.6 | grep -A40 "<__libc_start_main>"
```

Representative behavior:

```
__libc_start_main:
    ...
    call    __libc_init_first
    ...
    call    __libc_start_call_main
```

Responsibilities at this stage:

- Initialize glibc internal subsystems (threading, locale, memory allocator).

- Register process-wide atexit handlers.
- Prepare system interface fast-paths (VDSO dispatch, CPU feature path selection).

No C++ object construction occurs yet.

2.5.3 `__libc_start_call_main`: Invocation of C++ Static Initializers

Disassembly:

```
objdump -d /lib/x86_64-linux-gnu/libc.so.6 | grep -A40 "<__libc_start_call_main>"
```

Representative logic:

```
    call   __libc_csu_init   ; Execute .init_array constructors
    call   main              ; Transfer control to user code
    call   __libc_csu_fini   ; Registered for call at process exit
```

The important event is **execution of `.init_array`**. This array is generated by the toolchain and contains pointers to static-storage-duration constructors defined across all object files and linked libraries.
This confirms:

- C++ static objects are constructed **before** `main()`.
- The order is determined by ELF link order and `init_array` sequencing, not source order.

2.5.4 Validation of Constructor Execution Ordering

For a program containing:

```
#include <iostream>
struct X { X() { std::cout << "init\n"; } } x;
int main() {}
```

Execution order:

1. `_start` runs (loader entry).
2. `__libc_start_main` configures runtime.
3. `__libc_start_call_main` calls `__libc_csu_init`.
4. `X::X()` executes.
5. `main()` executes.

Thus, constructor execution precedes any user logic.

2.5.5 Summary of Verified Invariants

Stage	Symbol	Purpose	Visible Effect
Entry	`_start`	Establish ABI-compliant execution entry	Stack alignment and parameter forwarding

Stage	Symbol	Purpose	Visible Effect
Runtime Setup	`__libc_start_main`	Load and initialize runtime environment	Thread model, allocator, signal state
Static Initialization	`__libc_start_call_ma` and `.init_array`	Construct static objects, install destructors	Enables C++ object lifetime correctness
User Code	`main`	Program-defined logic execution	Normal execution semantics

The compiler generates code assuming these transitions are performed exactly in this order.

Any debugging, profiling, or runtime analysis must begin **before `main()`** to observe complete behavior.

Chapter 3

Toolchain Component Topology and Internal Data Flow

3.1 GCC → as → ld → ld.so → glibc → Application

The GNU compilation and execution pipeline is a staged transformation sequence, where each component contributes a distinct and non-overlapping function. The pipeline ensures that C++ source code is transformed into an executable binary that conforms to ABI rules and runtime initialization contracts.
The system model is:

```
GCC (frontend + optimizer + backend)
     ↓
as (assembler)
     ↓
ld (static/dynamic linker)
     ↓
ld.so (runtime dynamic loader)
```

```
    ↓
glibc (runtime environment and C++ initialization)
    ↓
Application code execution
```

The boundaries between these components define the **formal interfaces of program construction and execution**.

3.1.1 GCC: Language Semantics → Machine-Oriented IR → Assembly

GCC performs:

- Source parsing and semantic resolution (templates, overloads, concepts, constexpr evaluation).
- Transformation to GIMPLE SSA and mid-end optimization passes.
- Lowering to RTL and instruction selection according to target microarchitecture constraints.
- Register allocation and final scheduling.

Output is **assembly with relocation directives**, not a binary:

```
.text, .data, .bss sections + relocation records + symbolic references
```

GCC does not resolve external symbols or assign load addresses.

3.1.2 as: Assembly Encoding into ELF Relocatable Objects

The GNU assembler translates symbolic assembly to machine code and relocatable ELF objects:

- Emits `.text` with encoded instructions.
- Emits `.rela.text` / `.rel.text` entries describing unresolved symbol references.
- Defines symbol table entries (`.symtab`, `.dynsym`) and section boundaries.

No optimization occurs here.
`as` is a deterministic encoder; it does not transform semantics.

3.1.3 ld: Symbol Resolution, Address Assignment, and Relocation Planning

The linker (`ld` or gold/lld) constructs a complete ELF binary by:

- Resolving symbol definitions across object files and shared libraries.
- Constructing the **Global Offset Table (GOT)** for external data references.
- Emitting **Procedure Linkage Table (PLT)** stubs for deferred function resolution.
- Assigning segment load addresses (or emitting relocation tables in PIE builds).
- Assembling `.init_array` and `.fini_array` for constructor/destructor execution order.

If Link-Time Optimization (LTO) is enabled, GCC mid-end optimization may be re-entered before final object emission.
Without LTO, ld performs **no code optimization**, only structural linking.

3.1.4 ld.so: Runtime Relocation and Execution Environment Realization

For dynamically linked binaries, the kernel transfers control to **ld.so** at process start. `ld.so`:

- Maps shared libraries and resolves DT_NEEDED dependencies.
- Applies relocations generated by ld via `.rela.dyn` and `.rela.plt`.
- Finalizes GOT/PLT pointers.
- Initializes Thread-Local Storage structures.
- Establishes VDSO access paths using auxiliary vector information.

Only after these operations is the runtime environment ready for C/C++ initialization.

3.1.5 glibc: Runtime Subsystem Activation and C++ Static Object Initialization

glibc performs:

- Locale, malloc, and threading subsystem initialization.
- Registration of process-termination callbacks.
- Execution of constructors in `.init_array` (C++ static-storage-duration initialization).
- Transfer of control to `main`.

This phase ensures:

- ABI-consistent exception handling.
- Valid TLS descriptors.
- Deterministic constructor ordering.

3.1.6 Application: Execution Under Compiler-Defined Semantics

When `main()` executes:

- Control flow, data movement, and memory ordering follow GCC's lowered IR and scheduling decisions.
- All symbol bindings and runtime invariants are already established.
- Execution behavior is now entirely governed by the generated machine code and the C++ memory model.

3.1.7 Summary

Component	Responsibility	Defines
GCC	Semantic reduction and code generation	Program meaning
as	Instruction encoding and relocatable ELF formation	Binary representation
ld	Symbol resolution and layout	Link-time identity and visibility

Component	Responsibility	Defines
ld.so	Runtime relocation and environment realization	Dynamic execution state
glibc	Runtime and C++ initialization	Execution invariants
Application	Program logic execution	Observable behavior

The **system as a whole**, not the source file alone, defines the executable semantics of a C++ program.

3.2 Where Optimization Happens and Where It Cannot

Optimization in the GNU compilation pipeline is constrained by representation boundaries. GCC performs semantic and structural optimizations while the program is represented in compiler-owned intermediate forms. Once the program is lowered to assembly and subsequently linked, only limited structural transformations remain possible. Understanding these boundaries is essential for performance reasoning, binary reproducibility, and ABI stability.
The optimization landscape is divided into three distinct levels:

```
High-Level Language Semantics (C++ → GIMPLE)
      ↓
Target-Aware Intermediate Representation (GIMPLE → RTL)
      ↓
Machine Encoding (Assembly → ELF Object → Linked Binary)
```

Only the first two levels permit transformation of program semantics.

3.2.1 Optimization in the High-Level SSA Domain (GIMPLE)

GCC represents program flow and data dependencies in **GIMPLE in SSA form**, where:

- Side effects are isolated.
- Value propagation is explicit.
- Control paths are normalized.

Typical optimizations performed here include:

- Constant folding and propagation
- Dead code elimination
- Inlining based on cost modeling
- Loop interchange, unswitching, unrolling, and induction variable simplification
- Scalar replacement of aggregates
- Devirtualization when type identity is provable
- Forward substitution and redundant load elimination under alias constraints

These optimizations **preserve abstract semantics**. ABI-visible structure is not yet committed.

3.2.2 Optimization in the Machine-Constraint Domain (RTL)

After lowering to RTL (Register Transfer Language), code generation decisions are constrained by:

- Available instruction set architecture (ISA)
- Register allocation pressure
- Scheduling and port utilization characteristics
- Stack frame layout constraints
- Calling convention invariants

Optimizations here include:

- Instruction selection based on pattern matching

- Peephole folding and instruction substitution
- Register coalescing and allocation under interference constraints
- Basic block reordering for branch prediction and cache locality
- Scheduling for microarchitectural throughput and latency balance

At this stage, the compiler is limited to transformations that **do not change externally observable ABI properties**.

3.2.3 Where Optimization Cannot Occur: Assembler and Linker Phases

Once GCC emits assembly:

- The assembler (`as`) **does not optimize**. It performs byte encoding and relocation table construction only.
- The linker (`ld`) **cannot rewrite instruction streams**. It resolves symbol addresses and lays out sections according to ELF rules.

Exceptions:

- **Link-Time Optimization (LTO)** re-enters the GIMPLE/RTL pipeline before final linking, but only when explicitly enabled.
- `--icf=safe` and related identical code folding options operate on entire functions and cannot modify control flow granularity.
- **PLT/GOT resolution** affects dynamic dispatch cost but does not alter machine code sequencing.

Once an ELF binary is produced, **semantic structure is fixed**.

3.2.4 Where Optimization Is Explicitly Prohibited

Certain boundaries are required to preserve correctness:

Boundary Type	Reason Optimization Is Prohibited
External linkage call sites	Interposition possibility; correctness requires PLT calls
Functions exposed in shared libraries	Inlining would break binary compatibility
`volatile` memory accesses	Must be preserved in execution order
System call sites (`syscall`)	Privilege boundary enforces strict ordering
Atomic and synchronization primitives	Must comply with C++ memory model guarantees
Exception-handling tables (`.eh_frame`)	Structural stability required for unwinding

These boundaries are **visible in the generated assembly**. Performance analysis must account for them explicitly.

3.2.5 Engineering Consequence

Correct performance reasoning requires:

- Examining GIMPLE and RTL for transformation opportunities.
- Recognizing ABI constraints that prohibit optimization.
- Understanding that assembly and linking do not improve code quality.

- Using explicit compiler directives (`-fno-semantic-interposition`, `-fvisibility=hidden`, `-march`, `-mtune`) to control optimization availability.

The compiler is the **only layer** performing semantic optimization.
All subsequent toolchain stages **preserve**, rather than transform, the generated behavior.

3.3 How Debug Symbols Propagate Through the Pipeline

Debug symbols represent a structured mapping between **source-level program entities** and their **corresponding machine-level representations** in the final binary. Their propagation through the compilation and linking pipeline is governed by DWARF (Debugging With Arbitrary Record Formats) specifications and ELF section semantics. The compiler, assembler, and linker preserve these symbols unless explicitly removed, transformed, or stripped.
Debug information is **metadata**, not executable code; however, its correctness is required for stack unwinding, exception diagnostics, symbol resolution, profiling, and post-mortem analysis.

3.3.1 GCC: Generation of DWARF Symbol Information

When invoked with `-g`, GCC emits DWARF information during code generation. The emitted debug data describes:

- Compilation unit boundaries (`.debug_info`)
- Type definitions and C++ class layout metadata
- Source file and line correlation (`.debug_line`)
- Local and global variable scopes
- Inlined function expansion records
- Register and stack location expressions for variable reconstruction

This information is emitted into **parallel ELF sections**, not interleaved with executable code.
Example sections:

```
.debug_info
.debug_abbrev
.debug_line
.debug_ranges
.debug_str
```

The compiler does **not** commit frame pointer policies at this stage; unwinding behavior is described later.

3.3.2 as: Preservation Without Semantic Modification

The assembler (`as`) copies the DWARF sections verbatim into the relocatable object (`.o`) and generates the necessary **relocation entries** for debug symbols referring to code or data addresses. `as` does not generate, transform, or optimize DWARF. Its role is strictly:

- Encode instructions into `.text`
- Preserve debug metadata sections unchanged
- Emit relocation fixups linking debug records to symbol table entries

Thus, DWARF propagation through `as` is structurally transparent.

3.3.3 ld: Relocation, Folding, and Consolidation of Debug Sections

The linker (`ld`) performs the first non-trivial modification to debug data:

1. **Relocation resolution** updates debug metadata to point to final symbol addresses.
2. **Section merging** coalesces `.debug_*` sections across object files into unified segments.
3. **Dead-code elimination effects** may cause referenced debug records to become unreachable.
4. **Function inlining and identical code folding (ICF)** require remapping of PC ranges to source locations.

The linker does **not** discard debug symbols unless:

- `-s` (strip all symbols) or
- `--strip-debug` (strip debug-only sections)

is specified, or the link is performed under **separate debug info** mode.

3.3.4 Handling of Unwind Metadata

Unwind information is stored separately in:

- `.eh_frame`
- `.eh_frame_hdr`

Unlike general debug data, **unwind metadata is always retained**, even in stripped binaries, because:

- It is required for C++ exception propagation.

- It is used by backtrace and diagnostic subsystems.
- Runtime correctness depends on its availability.

This distinction is critical:
`.eh_frame` is execution-relevant; `.debug_\*` is not.

3.3.5 Separate Debug Information Model

Post-2020 distributions increasingly enable split debug info by default:

```
g++ -g -fdebug-prefix-map -gsplit-dwarf
```

This produces:

- Minimal debug metadata in the binary (compact representation).
- Full symbolic and type debugging information in `.dwo` or `.debug` files.

Dynamic debugging tools (gdb, perf, systemd-coredump) locate the external debug store using:

- `.gnu_debuglink`
- `.note.gnu.build-id`

This model improves:

- Cache locality (smaller runtime pages)
- Distribution and packaging efficiency
- Reproducibility under build system path normalization

3.3.6 Debug Symbol Visibility in Final Execution State

Stage	Debug Data Status	Notes
GCC	Full DWARF generated	Accurate source→IR mapping
as	Preserved	Relocation fixups applied
ld (normal link)	Unified and relocated	Unwind info retained regardless
ld (with split DWARF)	Debug data emitted into separate files	Binary only contains minimal tables
Stripped binary	Only `.eh_frame` retained	Still exception-safe but non-debuggable

Debug information is therefore part of the **compilation pipeline state**, not the executable semantics.
It must be treated as **first-class diagnostic infrastructure** in system-level C++ development.

3.4 How the Loader Chooses and Resolves Libraries

Dynamic linking on Linux x86-64 is governed by the ELF dynamic linking model. The loader (`ld.so`) resolves shared libraries required by the executable, establishes their dependency graph, binds symbols across shared objects, and finalizes the address and visibility of external references. This resolution process determines the **effective binary interface** and therefore defines which implementations of functions, symbols, and runtime facilities the application executes.
This makes the loader a **policy authority** for symbol binding, not a passive relocation executor.

3.4.1 Library Selection Process

The loader receives the runtime environment from the kernel, including argument vectors and the **auxiliary vector (auxv)**. Resolution of shared libraries follows this sequence:

1. **DT_NEEDED dependency list** embedded in the executable and shared objects.
2. **LD_LIBRARY_PATH** search paths (if permitted and not restricted by security policies).
3. Standard library directories (e.g., `/lib`, `/usr/lib`, multiarch paths).
4. Paths encoded using `RUNPATH` or `RPATH` ELF tags.

The actual lookup order is:

```
Executable
  → DT_NEEDED dependencies (in link order)
```

```
→ LD_PRELOAD overrides
→ Standard library paths (configurable via ld.so.cache)
```

Ordering is deterministic; the application cannot influence resolution at runtime except through `dlopen`-based dynamic loading.

3.4.2 DT_NEEDED and Dependency Graph Construction

Each shared object contains a `DT_NEEDED` entry listing libraries required to satisfy undefined references. The loader constructs a **directed dependency graph** and performs a topological traversal to ensure:

- Libraries are mapped exactly once.
- Their initialization routines run in correct dependency order.
- Circular references are resolved via lazy symbol binding rules.

This ensures consistency even in the presence of complex transitive link dependencies.

3.4.3 Symbol Lookup Scope and Resolution Rules

Symbol resolution follows strict ELF scoping semantics:

Scope Level	Resolution Behavior
Executable	Has highest priority for non-hidden symbols
Global Namespace	Libraries loaded in dependency order contribute visible symbols
Interposition	Symbols can be replaced by earlier scope unless visibility prevents it

Scope Level	Resolution Behavior
Hidden Symbols	Cannot be interposed; enforced at static link time

The compiler influences this through:

- `-fvisibility=hidden` (restrict interposition)
- `-fno-semantic-interposition` (allow inlining across DSOs)
- Linker scripts and symbol versioning blocks

The loader does **not** modify code generation; it enforces **binding policy** that determines whether indirect calls become PLT lookups or direct calls.

3.4.4 Lazy vs. Immediate Resolution

Procedure Linkage Table (PLT) entries may be bound:

- **Lazily** on first call (`LD_BIND_NOW` not set)
- **Immediately** at load time (`LD_BIND_NOW=1`)

Lazy binding reduces startup time but introduces:

- Runtime trampolines
- Unpredictable first-call latency
- Additional indirect branch cost

Immediate binding yields:

- Deterministic startup overhead

- Fully resolved GOT/PLT tables prior to execution
- More stable performance under profiling and latency-sensitive workloads

Modern high-performance deployments typically enable **immediate binding** explicitly.

3.4.5 Versioned Symbols and Compatibility Stability

glibc and other system libraries expose **versioned symbols** to ensure backward compatibility across releases. A single function name may reference multiple symbol versions:

```
memcpy@GLIBC_2.2.5
memcpy@GLIBC_2.14
```

The loader selects the version required by the executable's symbol table. This allows:

- Forward evolution of libraries
- Binary compatibility preservation
- Runtime coexistence of multiple ABI revisions

The compiler emits versioned symbol references automatically based on the headers and library versions detected at compile time.

3.4.6 Summary

Phase	Loader Function	Resulting Contract
Dependency Mapping	Reads `DT_NEEDED` and search paths	Establishes binary composition
Symbol Resolution	Applies ELF scoping and interposition	Determines visible interfaces
Relocation	Writes GOT/PLT and data references	Fixes execution binding topology
TLS and Initialization	Applies dynamic runtime configuration	Ensures ABI and memory model correctness

The loader finalizes **where and how** externally visible program behavior is implemented.
Its resolution rules therefore form part of the **effective execution semantics** of any C++ program on Linux.

3.5 Examples: Full Symbol Resolution Trace for a Shared C++ Binary

This section demonstrates the **symbol resolution process** for a dynamically linked C++ binary, showing how the loader establishes the final binding of function references across the executable, shared libraries, PLT/GOT dispatch points, and versioned glibc symbols. The objective is to expose the **runtime-visible dependency topology** generated during compilation and linking.
The example is intentionally simple but contains:

- A user-defined symbol with external linkage.
- A standard library symbol (`std::cout`).
- A glibc dependency resolved via the C++ runtime.

3.5.1 Source: Shared Library and Executable

Library (`libcalc.cpp`):

```
// libcalc.cpp
double add(double a, double b) { return a + b; }
```

Compile as shared library:

```
g++ -O2 -fPIC -shared libcalc.cpp -o libcalc.so
```

Executable (`main.cpp`):

```
#include <iostream>

extern double add(double, double);
```

```
int main() {
    std::cout << add(3.0, 4.0) << "\n";
}
```

Compile and link dynamically:

```
g++ -O2 main.cpp -L. -lcalc -o app
```

3.5.2 Inspecting Dynamic Dependency Graph

```
ldd ./app
```

Representative output:

```
libcalc.so => ./libcalc.so (0x00007f...)
libstdc++.so.6 => /usr/lib/... (0x00007f...)
libm.so.6 => /usr/lib/... (0x00007f...)
libgcc_s.so.1 => /usr/lib/... (0x00007f...)
libc.so.6 => /usr/lib/... (0x00007f...)
ld-linux-x86-64.so.2 => /lib64/... (0x00007f...)
```

This defines the **runtime search order** used by `ld.so`.

3.5.3 Symbol Resolution Trace Using `LD_DEBUG`

```
LD_DEBUG=libs,bindings ./app 2>&1 | less
```

Representative log excerpts:

```
calling init: /lib64/ld-linux-x86-64.so.2
calling init: ./libcalc.so
symbol=add;  lookup in ./app            => not found
symbol=add;  lookup in ./libcalc.so     => found: 0x00007f...
```

```
symbol=_ZSt4cout; lookup in ./app       => not found
symbol=_ZSt4cout; lookup in ./libcalc.so=> not found
symbol=_ZSt4cout; lookup in libstdc++.so.6 => found
symbol=__printf_chk; lookup in libc.so.6 => found
```

This confirms:

Symbol	Resolved In	Reason
`add`	`libcalc.so`	Exported by library with global visibility
`std::cout`	`libstdc++.so.6`	Standard library global symbol
`printf`-related internals	`libc.so.6`	I/O formatting backend used by stream output

The loader traverses dependencies in **deterministic link-time order**, not call-time order.

3.5.4 PLT/GOT Binding Inspection

```
objdump -d app | grep plt -A3
```

Representative result:

```
0000000000001080 <add@plt>:
    jmpq   *0x200a(%rip)     # GOT entry
    pushq  $0x0
    jmpq   0x1030            # PLT resolver stub
```

This confirms:

- Calls to `add` are **indirect through PLT** when semantic interposition is permitted.
- If compiled with `-fno-semantic-interposition -fvisibility=hidden`, the call site may be converted to a **direct relocation**, enabling inlining and constant propagation.

3.5.5 Versioned glibc Symbol Resolution

```
objdump -T /usr/lib/libc.so.6 | grep memcpy
```

Representative symbol table:

```
00000000000xxxx T memcpy@GLIBC_2.2.5
00000000000yyyy T memcpy@GLIBC_2.14
```

To determine which version is linked:

```
readelf -s app | grep memcpy
```

The symbol version record determines **which semantic contract** is used at runtime.

3.5.6 Summary of Verified Resolution Behavior

Mechanism	Responsible Component	Result
Symbol lookup order	`ld.so`	Deterministic dependency-directed traversal

Mechanism	Responsible Component	Result
Function indirection	PLT/GOT	Enables dynamic relocation and interposition
Library versioning	glibc symbol version tables	Preserves backward ABI compatibility
Final symbol binding	Loader relocation pass	Defines runtime execution behavior

Therefore, the **observable call graph** of a C++ program is a composite of:

- Compiler inlining decisions
- Link-time dependency resolution
- Loader binding policy
- Library version availability

It cannot be inferred from source code alone.

Part II

GCC FRONTEND: C++ LANGUAGE LOWERING ENGINE

Chapter 4

C++ Name Semantics, Lookup, and Instantiation Model

4.1 Unqualified, ADL, and Two-Phase Name Lookup

Name lookup in C++ determines which entities are associated with identifiers appearing in expressions. The correctness and meaning of a program depend on the precise rules for name resolution, particularly when templates and overload resolution interact. GCC implements the C++ lookup rules as defined by the standard's **two-phase name lookup model**, where lookup is separated into:

1. **Parsing and Template Definition Phase** (dependent names unresolved)
2. **Template Instantiation Phase** (dependent names resolved with context)

The compiler must determine which declarations are visible at each stage, while preserving the language's rules regarding scopes, namespaces, and argument-dependent lookup (ADL).

4.1.1 Unqualified Name Lookup

Unqualified name lookup applies when a name is referenced without an explicit scope qualifier. GCC resolves such names using a hierarchical search:

1. **Current block scope** (including local variables and parameters)
2. **Enclosing lexical scopes**
3. **Class scope** (for member function bodies)
4. **Namespace scope**
5. **Global scope**

Lookup stops at the **first scope level where a match is found**, regardless of whether overload resolution will later discard some candidates. This ensures deterministic resolution ordering independent of later semantic refinement. Unqualified lookup does **not** consider function arguments. Functions introduced later in the translation unit or by unrelated namespaces are not considered.

4.1.2 Argument-Dependent Lookup (ADL)

ADL supplements unqualified lookup by adding additional candidate functions based on the **types of function call arguments**. For each argument type:

- If the type belongs to a namespace, that namespace is added to the lookup set.
- If the type is a class, associated namespaces include:
  - The namespace containing the class definition
  - Namespaces of its base classes

- Namespaces of its member types used in operator overloads

Example:

```
namespace A { struct X {}; void f(X); }
void g() { A::X x; f(x); } // ADL finds A::f
```

Here, `f` is not found via unqualified lookup but is introduced through ADL.
ADL applies only to **function calls and operator syntax**, not variable or type lookup.

4.1.3 Two-Phase Name Lookup in Template Contexts

Two-phase lookup ensures correct resolution of names appearing inside templates that depend on template parameters.

1. **Phase 1 (Template Definition Time)**
   - Non-dependent names are resolved immediately using standard lookup.
   - Dependent names remain unresolved placeholders.
2. **Phase 2 (Template Instantiation Time)**
   - Dependent names are resolved after substituting template arguments.
   - ADL applies only at instantiation, based on concrete argument types.

Example:

```
template<typename T>
void h(T t) { f(t); } // f is dependent; not resolved yet
```

Only when `h<int>` is instantiated does GCC determine which `f` to invoke, and ADL applies based on the type `int`.
This model prevents accidental binding of template bodies to unrelated declarations visible only at definition time.

4.1.4 Failure Modes and GCC Diagnostic Behavior

Two categories of lookup failures exist:

Failure Type	Occurs When	Diagnostic Timing
Non-dependent name not found	Not visible during template definition	Error during parsing
Dependent name not resolved at instantiation	No valid candidates after substitution and ADL	Error during instantiation

This distinction ensures program correctness is enforced **declaratively and incrementally**, aligned with template specialization semantics.

4.1.5 Practical Implications for System-Level C++ Development

- Namespace partitioning must be intentional; accidental ADL effects can alter overload resolution.
- Inline namespaces and library versioning require explicit visibility control to prevent unintended lookup expansion.
- Generic code correctness depends on understanding when a name is dependent; failure to do so results in silent ambiguity or late-stage instantiation errors.
- GCC's template instantiation engine assumes two-phase lookup strictly; optimizations and overload narrowing cannot occur before substitution.

4.1.6 Summary

Lookup Mechanism	**Trigger Condition**	**Scope Determination**	**Resolution Timing**
Unqualified Lookup	Identifier without qualifier	Lexical scope hierarchy	Parse time
ADL	Function call expressions	Namespaces associated with argument types	Instantiation time (if dependent)
Two-Phase Lookup	Template contexts	Combination of above depending on dependency	Split across definition and instantiation

Name resolution in GCC is therefore a **semantic reduction process**, not a lexical matching algorithm.
Its correctness determines the **structural meaning of expressions**, and therefore the **generated machine code semantics**.

4.2 Template Pattern Matching and Partial Specialization Ordering

Template specialization resolution in C++ is a **pattern matching and order selection problem** performed by the compiler during instantiation. GCC's template instantiation engine must identify which specialization, if any, matches the supplied template arguments, and then select the *most specialized* viable declaration according to a strict partial ordering relation defined by the standard.
This resolution mechanism is essential for generic programming, metaprogramming, and the behavior of standard library components such as type traits, iterator adapters, and allocator dispatch layers.

4.2.1 Primary Templates and Explicit Specializations

A **primary template** defines a general pattern:

```
template<class T>
struct S { /* generic case */ };
```

An **explicit specialization** replaces the primary template entirely for a specific argument set:

```
template<>
struct S<int> { /* specialized case */ };
```

Explicit specializations bypass pattern matching and are selected by exact type identity comparison.

4.2.2 Partial Specializations and Pattern Matching

A **partial specialization** constrains the primary template's pattern:

```
template<class T>
struct S<T*> { /* pointer type case */ };
```

During instantiation, GCC matches the specialization pattern to the template argument list, performing:

1. **Type structure decomposition** (matching structure, not just names).
2. **Bidirectional type substitution checks** to determine if the specialization is applicable.
3. **Deduction of template parameters** based on pattern positions.

Matching fails if:

- Type shapes differ (e.g., pointer vs non-pointer).
- Required parameter deduction is ambiguous.
- Constraints (e.g., requires-clauses) evaluate to false.

4.2.3 Partial Ordering: Determining the Most Specialized Match

When multiple partial specializations match, GCC determines which one is **more specialized**.
The rule: *A specialization A is more specialized than specialization B if A can be used to instantiate B, but B cannot be used to instantiate A.*
Example:

```
template<class T>
struct S<T*>;        // (1) pointer specialization

template<class T>
struct S<T* const>;  // (2) const pointer specialization
```

For an instantiation `S<int* const>`:

- Both (1) and (2) match.
- Substituting (2)'s pattern into (1) succeeds, but the reverse does not.
- Therefore, specialization (2) is **more specialized**.

The compiler's partial ordering algorithm is **structural**, not textual or semantic.

4.2.4 Interaction with Function Template Partial Specialization

Function templates do **not** support partial specializations. Instead, they rely on:

- **Overload resolution**
- **Template argument deduction**
- **Constraint-based selection (post-C++20 concepts)**

Example:

```
template<class T>
void f(T);

template<class T>
requires std::is_integral_v<T>
void f(T);
```

Here, the constrained version is selected by constraint satisfaction, not pattern ordering. This distinction becomes critical when reasoning about overload resolution in generic code.

4.2.5 Constraint-Based Ordering (C++20 Concepts)

Post-C++20, GCC evaluates **requires-clauses** and **concept constraints** as part of the ordering relation.
Given two viable function or class template overloads:

- The one with the **more constrained** requirement set is preferred.
- Constraint subsumption replaces earlier SFINAE-based partial ordering.

Example:

```
template<class T>
concept Iterable = /* detection logic */;

template<Iterable T>
void g(T);            // selected for iterable types

template<class T>
void g(T);            // fallback
```

The compiler selects `g(T)` with the `Iterable` constraint when satisfied, without ambiguity.

4.2.6 Failure Modes and GCC Diagnostic Context

Failure Mode	Cause	Diagnostic Behavior
No matching specialization	Pattern mismatch	Reported at instantiation
Multiple equally specialized matches	Ambiguous partial ordering	Hard error
Constraint evaluation failure	`requires` evaluates to false	Candidate removed prior to overload resolution
Recursive template selection loops	Indirect specialization referencing	Diagnosed via instantiation depth limits

GCC reports instantiation traces with backreferences to template points of definition to facilitate debugging of metaprogramming logic.

4.2.7 Summary

Mechanism	Scope	Solver	Order Rule
Explicit Specialization	Exact argument match	Identity comparison	No ordering needed
Partial Specialization	Template structural pattern matching	Type decomposition + deduction	Most specialized wins
Function Templates	No partial specialization; overload instead	Standard overload resolution	More constrained signature wins

Mechanism	**Scope**	**Solver**	**Order Rule**
Constraints/Concepts (C++20+)	Semantic filtering of candidates	Constraint evaluation system	Constraint subsumption ordering

Template specialization resolution is therefore a **formal selection and ordering problem**, not an ad-hoc name match.
Its correctness is foundational to **generic C++ and standard library behavior**.

4.3 Constraint Subsumption Rules in Concepts

Constraint subsumption defines the selection ordering between overloaded templates whose viability is determined by **concept constraints** rather than structural template argument matching. It replaces earlier SFINAE-based ordering logic with a **well-defined partial order** over constraint expressions. This mechanism ensures deterministic overload resolution in generic code that depends on semantic conditions rather than syntactic pattern matches.

4.3.1 Constraint Normalization

Before evaluating subsumption, constraints are transformed into a **normalized canonical form**.
Normalization includes:

- Expansion of composite constraints (`&&`, `||`) into structured predicate sequences.
- Replacement of abbreviated function templates with explicit constraint expressions.
- Evaluation of trivially true or syntactically redundant constraints.

GCC performs normalization at template definition time, allowing later instantiation to rely solely on constraint evaluation rather than structural rewriting.

4.3.2 Constraint Implication and Subsumption

Given two viable constrained templates `A` and `B`, `A` is **more constrained** than `B` if:

```
For all template arguments where A is satisfied,
B is also satisfied,
and there exists at least one argument set where B is satisfied but A is not.
```

Symbolically:

```
A   B   and   B   A
→ A is more constrained than B
```

Subsumption establishes the ordering used by overload resolution:

- If one constraint implies the other, the more restrictive template is preferred.
- If implication is bidirectional (logical equivalence), ordering is ambiguous → diagnostic error.
- If neither implies the other, the templates are incomparable → overload ambiguity.

4.3.3 Example: Ordered Constraints

```
template<class T>
concept Integral = std::is_integral_v<T>;

template<class T>
concept SignedIntegral = Integral<T> && std::is_signed_v<T>;

void f(Integral auto);        // (1)
void f(SignedIntegral auto);  // (2)
```

Constraint implication:

```
SignedIntegral(T)   Integral(T)
Integral(T)   SignedIntegral(T)
```

Therefore (2) **subsumes** (1).
A call `f(-1)` selects **(2)**.

4.3.4 Example: Incomparable Constraints

```
template<class T>
concept Floating = std::is_floating_point_v<T>;

template<class T>
concept Bounded = requires { typename T::bounds; };

void g(Floating auto);   // (1)
void g(Bounded auto);    // (2)
```

Here:

- `Floating(T)` does not imply `Bounded(T)`
- `Bounded(T)` does not imply `Floating(T)`

For a type satisfying both, overload resolution **fails** due to lack of subsumption ordering.
GCC emits an ambiguity diagnostic at instantiation.

4.3.5 Interaction with Function Overload Resolution

Constraint subsumption is applied **before** parameter-dependent overload resolution:

1. Filter candidates by constraints (discard those failing).
2. Order remaining candidates by constraint subsumption.
3. Apply standard overload resolution among equally constrained candidates.

This ordering prevents accidental selection of weaker templates during overload selection, eliminating patterns previously handled by fragile SFINAE idioms.

4.3.6 Replacement of SFINAE-based Partial Ordering

Pre-C++20 generic libraries used SFINAE to express substitution failure as exclusion. Constraint subsumption provides:

Characteristic	**SFINAE**	**Concepts (>=C++20)**
Failure Mode	Substitution collapse	Logical constraint evaluation
Ordering	Emergent, indirect	Explicit through implication rules
Diagnostics	Late, complex	Early, explicit, context-rich
GCC Implementation	Template substitution engine	Constraint solver + semantic lattice

Concepts remove the implicit reliance on overload failure as a selection mechanism.

4.3.7 Summary

Mechanism	**Purpose**	**Ordering Rule**
Constraint Normalization	Establish canonical evaluation form	Applied at template definition
Subsumption	Determine “more constrained” overload	Logical implication A B
Overload Selection Integration	Replace SFINAE ordering	Constraint order precedes type-based overload rules

Constraint subsumption gives the compiler a **declarative and deterministic** method for resolving overloaded templates governed by semantic requirements, ensuring

stability and clarity in modern C++ generic code.

4.4 Pure Compile-Time Execution in `constexpr` Interpreter

`constexpr` evaluation establishes a semantic domain where expressions, functions, and object construction can be executed **entirely at compile time**, producing values that become part of the program image rather than runtime computation. GCC implements this through an internal **constant evaluation engine** that simulates execution on an abstract machine distinct from the actual target architecture. This interpreter enforces compile-time rules that guarantee referential transparency, memory safety within the evaluation domain, and deterministic behavior.

4.4.1 Execution Model: Abstract Machine for Constant Evaluation

During constant evaluation, code is executed in a **side-effect-restricted environment**:

- Only memory allocated in the interpreter's **constant object arena** may be accessed.
- Pointers refer only to interpreter-managed memory locations; no interaction with runtime storage is permitted.
- All control flow constructs are supported, but only operations that yield compile-time-deterministic results are valid.

The interpreter maintains:

- A virtual stack and activation record structure

- A compile-time heap for objects with static extent
- A symbolic representation of objects and their lifetimes

No actual machine registers or hardware execution occur.

4.4.2 Eligibility Rules for `constexpr` Evaluation

An expression is evaluated at compile time if:

1. The expression is required in a context that mandates a constant expression (e.g., array bound, template argument).
2. The function invoked is declared `constexpr` or `consteval`.
3. All operations within the expression are valid within the constant evaluation domain.

Violating operations include:

Operation Type	**Disallowed Reason**
Dynamic allocation (`new`, `malloc`)	Requires runtime-managed heap
I/O or OS interaction	Has external side effects
Non-literal type manipulation without `constexpr` constructors	Cannot form stable compile-time object graphs
Undefined behavior triggers	Compile-time interpreter enforces strict diagnostics

The interpreter performs **hard rejection** of invalid operations at compile time.

4.4.3 Persistent Object Representation at Compile Time

Objects created during constant evaluation fall into two classes:

1. **Immediate Constants**
   Represented directly in the expression graph; lowered to compile-time literals.

2. **Extended Object Graphs**
   Structured memory layouts representing class instances, arrays, and nested aggregates stored in the interpreter-managed arena.

When used in a context requiring actual code emission, GCC emits **static storage objects** into `.rodata` or into constant-propagated immediate operands.
Example:

```
constexpr int f() { return 6 * 7; }
static_assert(f() == 42);        // computed in compile-time domain
constexpr int x = f();           // stored as integer literal
```

No runtime call to `f()` appears in the linked binary.

4.4.4 Distinction Between `constexpr` and `consteval`

Keyword	Timing	Enforcement	Runtime Availability
`constexpr`	Compile-time preferred, runtime permitted	Conditional execution allowed	Yes
`consteval`	Must execute at compile time	Runtime execution forbidden	No

Example:

```
consteval int make() { return 5; }
constexpr int v = make(); // OK
int r = make();           // Compile-time error
```

`consteval` produces **compile-time-only functions**; GCC emits no callable runtime code.

4.4.5 Interaction with Template Instantiation

Constant evaluation may occur **before**, **during**, or **after** template instantiation depending on dependency:

- Non-dependent expressions are executed during template definition.
- Dependent expressions defer evaluation to instantiation.
- If the result becomes constant, GCC propagates it into subsequent optimization passes (constant folding, dead branch elimination, loop unrolling).

Constant evaluation therefore directly feeds **mid-end optimization**.

4.4.6 Engineering Significance

Compile-time evaluation provides:

- Reduced runtime cost (eliminating repeated computation).
- Deterministic initialization of static objects.

- Ability to construct complex lookup tables, state machines, or precomputed transforms.
- Increased guarantees of defined behavior (interpreter rejects UB at compile time).

However:

- Code intended for compile-time must avoid dependency on runtime resources.
- Excessive interpreter complexity increases compile-time cost.

4.4.7 Summary

Property	**Meaning**	**Enforced By**
Execution Domain	Abstract evaluator, not CPU	GCC constexpr interpreter
Side-Effect Restriction	Only pure, deterministic operations permitted	Compile-time semantic validation
Object Placement	Values lowered into `.rodata` or literals	Constant propagation and data emission
Language Interaction	Fully integrated with templates and overload resolution	Definition-time and instantiation-time evaluation

`constexpr` evaluation is therefore a **formal semantic execution environment** embedded within the compiler, not an optimization heuristic.

4.5 Examples: GCC AST Graph Analysis with `-fdump-tree-original-raw`

The GCC C++ frontend constructs an Abstract Syntax Tree (AST) representation before lowering to GIMPLE. The raw AST form retains complete syntactic structure, scope nesting, template argument bindings, unqualified name lookup results, and semantic annotations necessary for later phases. Using the dump facility:

```
g++ -fdump-tree-original-raw -c file.cpp
```

produces a textual AST graph before any semantic lowering or canonicalization steps. This provides direct visibility into how name lookup, template binding, and overload relationships are interpreted by the compiler at the earliest resolvable stage.

4.5.1 Example Source

```
#include <iostream>

template<class T>
T sqr(T x) { return x * x; }

int main() {
    int a = 7;
    std::cout << sqr(a) << "\n";
}
```

Compile:

```
g++ -O2 -fdump-tree-original-raw -c example.cpp
```

This generates:

```
example.cpp.003t.original
```

4.5.2 Relevant Dump Segments (Simplified for Presentation)

Excerpt:

```
@1 function_decl name: sqr
  type <@2>
  arguments: (x @3)
@2 function_type returns: T type@4
@3 parm_decl name: x type: T type@4

@4 template_type_parm index: 0 level: 0

@10 function_decl name: main
  body:
    {
      @11 var_decl name: a type: int
      @12 modify_expr
         lhs: a
         rhs: integer_cst 7

      @13 call_expr
         fn: @14
         args: @15
    }

@14 addr_expr of function_decl name: sqr   <-- template specialization resolved

@15 call_expr
     fn: overloaded_operator<<
     args: (cout, call_expr sqr(a))
```

This reveals:

- The **template parameter** `T` (`@4`) remains symbolic until instantiation.

- The call `sqr(a)` instantiates `T=int` during semantic checking, reflected at `@14`.
- Stream insertion uses overloaded operator resolution via namespace-scope lookup.

4.5.3 Observations on Name Resolution and Semantic Binding

Construct	AST Evidence	Interpretation
`sqr(a)`	`addr_expr` referencing `sqr`	Unqualified lookup resolved; template instantiation performed
Template parameter	`template_type_parm index: 0`	`T` is a deduced dependent parameter until instantiation
`std::cout` handling	Resolved through namespace `std`	Lookup occurred before GIMPLE transformation
Operator overload	`call_expr overloaded_operator<<`	ADL and overload resolution applied in AST phase

This verifies that name lookup and overload binding occur **prior to** lowering to GIMPLE SSA form.

4.5.4 Using AST Dumps for Diagnostic Analysis

Raw AST inspection is used to:

- Confirm that lookup resolves to the intended declaration.
- Detect unintended ADL visibility expansion.
- Validate template parameter deduction paths.

- Analyze overload resolution priority selection.
- Identify incorrect namespace qualification assumptions.

Especially in generic code, AST inspection reveals **whether the compiler matched the author's conceptual model** of name binding.

4.5.5 Limitations and Interpretation Boundaries

The raw AST is **not executable** and does **not** represent control-flow or data-flow.
It precedes:

- Early constant folding
- Template code canonicalization
- Inline function expansion
- SSA form generation

To analyze flow-sensitive behavior, one must inspect later dumps:

```
-fdump-tree-gimple
-fdump-tree-optimized
```

However, **binding correctness must always be verified at the AST level**.

4.5.6 Summary

Stage	Artifact	Purpose
AST (raw)	`.003t.original`	Name binding, template binding, overload selection
GIMPLE SSA	`.optimized` dumps	Control-flow and data-flow transformation domain
RTL / Scheduling	`.rtl` dumps	Target-specific instruction selection and allocation

`-fdump-tree-original-raw` is the **primary inspection point** for ensuring the correctness of name lookup, template specialization selection, and overload resolution semantics in GCC's C++ frontend.

Chapter 5

Semantic Graph to GIMPLE Transformation Pipeline

5.1 Canonicalization of Expressions and Control Flow

After parsing and semantic resolution, GCC lowers the C++ Abstract Syntax Tree (AST) to **GIMPLE**, an explicitly structured intermediate representation. The lowering process discards syntactic sugar and normalizes program structure into a form suitable for static single assignment (SSA) construction, control-flow optimization, and data-flow analysis. Canonicalization ensures that the IR expresses computation in **simple, explicit, and analyzable operations** with no implicit sequencing or context-dependent interpretation.

5.1.1 Expression Canonicalization

The C++ expression grammar allows compound constructs with operator overloading, implicit conversions, and sequencing rules. GIMPLE canonicalization eliminates these

forms by:

1. **Breaking complex expressions into three-address operations**, each with at most one operator.
2. **Materializing all intermediate values into temporaries**, ensuring explicit data dependencies.
3. **Lowering overloaded operators to function calls or intrinsic sequences**, based on semantic resolution.
4. **Eliminating implicit conversions**, replacing them with explicit cast operations where required.

Example transformation:
Source:

```
int y = (a + b) * f(x);
```

Canonical GIMPLE form (conceptual):

```
t1 = a + b;
t2 = f(x);
y = t1 * t2;
```

Every data dependency becomes explicit, enabling data-flow reasoning and SSA transformation.

5.1.2 Control-Flow Canonicalization

C++ control constructs (e.g., `for`, `while`, `if`, exception propagation) are normalized into a **minimal set of primitive control-flow structures**:

- Conditional and unconditional jumps
- Basic blocks forming a directed control-flow graph (CFG)
- Structured exception-handling edges represented via landing pads and unwind paths

High-level loops are converted into:

1. A loop header block (entry and iteration test point)
2. A body block
3. A backedge block returning to the header
4. A loop exit block

This canonical structure is necessary for:

- Loop-invariant code motion
- Induction variable analysis
- Bounds inference and vectorization preparation

5.1.3 Side-Effect Isolation

To enable formal reasoning and optimization, canonicalization separates effects from evaluation:

- Function calls are represented as **call statements** with explicit parameter evaluation order.

- Stores to memory are expressed as explicit store operations with typed destination regions.
- Volatile and atomic accesses are represented as non-reorderable statements with explicit ordering constraints.

Side-effect isolation is essential for determining:

- Alias relationships
- Expression reordering legality
- Dead-store and redundant-load elimination
- Memory-model-consistent parallel transformation

5.1.4 Exception Flow and the EH Graph

Exception semantics cannot be represented purely through CFG edges because they propagate non-locally. GCC introduces an **exception-handling graph (EH graph)**:

- Each potentially throwing instruction carries metadata describing its unwind target.
- The CFG and EH graph together define full program control flow.
- Cleanup regions and catch handlers become first-class dispatch nodes in the combined graph.

This separation enables:

- Precise modeling of destructors and RAII cleanup paths.
- Region-based elimination of unused exception edges when optimization proves non-throwing behavior.

5.1.5 Canonical Form Guarantees

The canonical GIMPLE form satisfies:

Property	Guarantee	Required For
Single-operation expressions	One operator per statement	SSA construction and value propagation
Explicit control edges	No implicit fallthrough semantics	CFG optimization and scheduling
Deterministic evaluation order	Sequence points resolved early	Side-effect analysis
Uniform loop representation	Canonical header-body-backedge-exit form	Loop optimization pipelines
Isolated exception regions	EH and CFG separation	Correct destructor and unwinding semantics

Canonicalization therefore converts **syntactic structure** into **formal execution structure**, enabling deterministic and analyzable transformation.

5.1.6 Summary

GIMPLE canonicalization:

- Removes syntactic complexity and implicit sequencing.
- Converts expressions and control flow into normalized, analyzable forms.
- Establishes the structural foundation required for SSA, constant propagation, alias analysis, loop optimization, vectorization, and instruction scheduling.

This transformation is the **semantic bridge** between high-level C++ constructs and the lower-level IR on which optimization and code generation operate.

5.2 Temporary Lifetime Folding and Value Category Lowering

C++ defines a rich value category model (lvalue, xvalue, prvalue), along with complex temporary object lifetime rules governing construction, destruction, and elision. Before optimization and SSA construction, GCC lowers these high-level semantics into **explicit object materialization and destruction operations** in GIMPLE. The objective is to remove implicit lifetime boundaries and ensure that all temporaries are represented as **storage-backed entities** or **pure values** depending on their usage context. This transformation is referred to as **temporary lifetime folding**.

5.2.1 Value Category Normalization

The C++ expression model distinguishes:

Category	Meaning	Lowered Representation
prvalue	Pure value, no identity	Scalars: SSA values; Classes: forced materialization into temporary storage
xvalue	Expiring object	Lvalue reference to a known storage region with move semantics
lvalue	Named or addressable object	Direct reference to an allocated region

During lowering:

1. **Scalar prvalues** become **SSA values** (no storage allocated).

2. **Class-type prvalues** become **temporary objects** with storage in a compiler-managed stack slot.

3. **xvalues** are treated as lvalues with explicit move operations inserted if needed.

This step establishes the **storage identity** of every non-scalar temporary.

5.2.2 Materialization Points and Temporary Storage Creation

C++ requires object materialization when a prvalue is used in a context requiring storage identity, including:

- Binding to a reference
- Passing as function arguments to parameters of reference type
- Producing subobjects via member access or operator selection

Example:

```
X make();
X y = make();
```

Lowered conceptual GIMPLE form:

```
t0 = make();          // materialize X object
y = t0;               // copy or move as permitted
```

If elision applies, the temporary slot becomes identical to the storage of `y`, folding the lifetime boundary.

5.2.3 Lifetime Folding and Elision

Under guaranteed copy elision (C++17 onward), temporary lifetimes may be **collapsed into the lifetime of the destination object**, eliminating intermediate construction:

```
return X();  // no temporary; directly constructs return object storage
```

GCC performs this during GIMPLE construction by:

- Unifying object allocation sites
- Eliminating intermediate storage objects
- Rewriting constructor calls to use final target memory

This yields a **single storage region**, reducing destructor scheduling and eliminating copies.

5.2.4 Destructor Scheduling and Region Boundaries

Temporary objects that remain materialized must have explicit lifetimes. GCC emits:

- `__builtin_lifetime_start` markers at allocation
- `__builtin_lifetime_end` markers at destruction
- Explicit destructor calls inserted at well-defined scope exit points

Example conceptual lowering:

```
{ X t = make(); use(t); } // block scope
```

becomes:

```
t = _materialize_X();
call X::X(&t, make());
use(t);
call X::~X(&t);
```

These boundaries are necessary for:

- Exception-safe unwinding tables
- Precise RAII cleanup sequencing
- Correct alias and escape analysis

5.2.5 Move/Copy Lowering and Value Propagation

During lowering, GCC distinguishes between:

- **Move-eligible** transfers: implemented via `X(X&&)` constructor
- **Copy-required** transfers: implemented via `X(const X&)`

The decision depends on value category normalization:

Expression Form	Lowered Operation
prvalue used to initialize object	Move or elide
xvalue passed to parameter	Move
lvalue passed to parameter	Copy

These choices affect aliasing and optimization viability downstream in GIMPLE SSA.

5.2.6 Result of Lifetime Folding Before SSA Form

After canonicalization:

- All temporary objects are either **elided** or **explicitly represented**.
- All construction and destruction operations are **structurally visible**.
- Value propagation flows between explicit SSA names or identifiable memory regions.
- There are **no implicit lifetime boundaries** remaining in the IR.

This explicit representation is mandatory before:

- SSA -node insertion
- Alias classification
- Escape and points-to analysis
- Loop and region-based optimizations

5.2.7 Summary

Transformation	Purpose	Resulting IR Property
Value category lowering	Remove abstract prvalue/xvalue distinctions	Explicit memory vs value representation
Temporary lifetime folding	Collapse storage where elision is permitted	Reduced allocations and destructor calls

Transformation	Purpose	Resulting IR Property
Materialization of class-type prvalues	Give identity to objects requiring storage	Clear alias and ownership semantics
Destructor scheduling insertion	Make object lifetime explicit	Correct RAII and unwinding semantics

Temporary lifetime folding ensures that **object identity, lifetime, ownership, and movement** are explicit in GIMPLE, enabling precise and correct optimization.

5.3 Lambda Closures, Captures, and Object Lifetime IR Representation

Lambda expressions in C++ are lowered into **closure objects** whose structure and semantics are fully determined during frontend AST canonicalization. GCC generates a class type representing the closure, synthesizes its data members corresponding to captured entities, and emits an overload for `operator()` implementing the lambda body. This transformation ensures that the callable entity is represented explicitly in GIMPLE, allowing standard object lifetime, aliasing, and optimization rules to apply.

5.3.1 Closure Type Synthesis

For each lambda expression, GCC constructs a unique, unnamed class type:

```
struct <lambda closure> {
    // data members for captures
    auto operator()(parameter-list) const;
};
```

Properties:

- The closure type has no user-visible identifier, but is fully represented in the AST.
- Closure types are non-aggregate unless all captures are public, trivial, and non-static (post-C++20 changes allow constexpr closures under expanded rules).
- If the lambda is marked `mutable`, the generated `operator()` is non-const.

This closure type is the unit of representation for capturing semantics in IR.

5.3.2 Capture Lowering and Storage Identity

Capture categories map directly to member field layout:

Capture Form	Representation	Notes
Capture by value (`[x]`)	Member field storing copy of `x`	Copy or move semantics applied during closure construction
Capture by reference (`[&x]`)	Member field holding pointer/reference to original `x`	No copy; lifetime dependency must be preserved
Capture of `this`	Member storing pointer to current object	Treated identically to `[this]` capture
Default captures (`[=]`, `[&]`)	For each referenced entity, apply default mode	Resolved at semantic analysis time

These captured members are treated as **ordinary data members** in the closure type; GIMPLE has no special-case representation for captured variables beyond standard memory fields.

5.3.3 Construction and Destruction of Closure Objects

Closure creation is lowered into explicit constructor-like initialization:
Example:

```
auto f = [x](int y) { return x + y; };
```

Lowered conceptual GIMPLE:

```
closure.temp = _closure_type(x);   // materialize closure object
```

For value captures, the closure stores a **copy** of the captured variable.
For reference captures, the closure stores an **address**, requiring no object duplication.
Destruction follows standard automatic storage lifetime rules; no special destructor is emitted unless captured types require destruction.

5.3.4 Lowering `operator()` and Call Sites

The lambda body is compiled as:

```
return closure.operator()(arguments);
```

In GIMPLE:

- `operator()` becomes a normal function with an implicit `this` pointer.
- References to captured entities become `MEM_REF`s to closure fields.
- Inline propagation applies when optimization is enabled, allowing closure elimination if storage does not escape.

Captured values are propagated like regular SSA variables unless address-taking prevents forwarding.

5.3.5 Escaped Closures and Heap Promotion

If the closure is returned, stored globally, or passed to another function, GCC determines that:

- The closure must be assigned stable storage (stack or heap).
- Captured references must preserve extended lifetime correctness.
- Alias and escape analysis are updated to reflect possible external use.

If escape is detected, closure fields become **observable state** and cannot be elided. Example of lifetime escape:

```
return [x]() { return x; }; // closure escapes caller frame
```

Here, `x` capture must follow copy semantics; reference capture would produce a dangling pointer.

5.3.6 Interaction with SSA and Optimization

Once lowered:

- Closure fields participate in SSA value propagation.
- Constant-captured values may be folded into immediate operands.
- Unused captures are removed by dead-field elimination.
- If the closure is inlinable and non-escaping, GCC may eliminate both the closure object and its `operator()` function entirely (closure flattening).

Optimization condition summary:

Closure Property	Optimization Result
Non-escaping + trivial captures	Closure elimination and call-site inlining
Escapes via return or storage	Closure persists as a first-class object
Captures references	Lifetime constraints prevent elimination

5.3.7 Summary

Transformation Stage	Closure Representation Outcome
AST Lowering	Closure type definition + `operator()` synthesis
Capture Analysis	Captured entities mapped to closure fields
Materialization	Closure objects constructed explicitly in GIMPLE
SSA/Optimization	Closures propagated, folded, or eliminated depending on escape and alias conditions

Lambda closure lowering ensures that **capturing semantics, object identity, and lifetime boundaries** are explicit and analyzable at the IR level.
Once represented in GIMPLE, closures behave uniformly with other objects, enabling full optimization under standard data-flow and aliasing models.

5.4 Inline and Devirtualization Decision Models at GIMPLE Level

Inlining and devirtualization are performed during GIMPLE-stage optimization. Both transformations replace indirect or out-of-line call sites with more direct call forms to improve instruction locality, remove call overhead, expose scalarization and constant propagation opportunities, and unlock further mid-end optimizations. Since these transformations change the shape of the control-flow graph and value propagation domain, they occur **after canonicalization** but **before SSA-based optimizations are finalized**.

5.4.1 Inlining Candidate Identification

A call is eligible for inlining if:

1. The callee body is available in the compilation unit (or via Link-Time Optimization).
2. The callee is not externally interposable (subject to symbol visibility and semantic interposition rules).
3. The callee's size and control complexity fall within inlining thresholds.

GCC determines eligibility using a cost model that evaluates:

- Instruction count
- Basic block count and loop depth
- Expected register pressure due to inlining

- Potential constant propagation benefits due to visible arguments
- Hotness feedback from `-fprofile-generate/-fprofile-use` or `-fauto-profile`

Inlining is performed to **enable optimization visibility**, not merely to remove call overhead.

5.4.2 Visibility and Interposition Constraints

Inlining may be legally blocked if a function is **interposable** through the dynamic linker. The compiler assumes a function is interposable unless:

- The function is declared `static`, or
- The symbol has hidden visibility (`-fvisibility=hidden`), or
- The binary is compiled with `-fno-semantic-interposition`.

Without these constraints, GCC cannot assume that the function definition visible at compile time is the one executed at runtime. In that case:

```
Call remains indirect through PLT → no inlining permitted.
```

Thus, symbol visibility configuration directly influences optimization capability.

5.4.3 Devirtualization Pre-Conditions

A virtual call:

```
obj->vfunc(args)
```

may be rewritten as a direct call if the compiler can prove that:

1. The dynamic type of `obj` is known, and

2. That type's vtable entry for `vfunc` is statically determined.

Proof sources include:

- Whole-object construction analysis
- Final class detection (`final` keyword or no derived class seen in linkage graph)
- Type propagation in SSA
- Devirtualization hints from profile data

If devirtualization succeeds:

```
obj->vfunc(args)
↓
(&ClassName::vfunc)(obj, args)
↓
direct call to function body
```

This eliminates vtable lookup and typically enables inlining.

5.4.4 GIMPLE-Level Transformation Form

Before inlining or devirtualization:

```
call obj->vfunc(args)
```

After devirtualization:

```
call ClassName::vfunc(obj, args)
```

After inlining:

```
<inlined function body inserted at call site>
```

Post-inlining, further optimizations apply:

- Constant folding of captured or propagated parameters
- Dead branch elimination inside inlined control flow
- Scalar replacement of aggregates

Inlining therefore **expands** optimization visibility scope.

5.4.5 Profile-Guided and Cost-Driven Inline Decisions

With PGO (`-fprofile-use`):

- Call frequency and basic block hotness weight the inliner's cost model.
- Hot functions and hot call edges in the call graph are inlined more aggressively.
- Cold call paths are left uninlined to reduce code growth.

Without profile data:

- Heuristics use instruction count thresholds, call depth limits, and structural scoring.

This makes inlining a **performance-critical and architecture-dependent** decision layer.

5.4.6 When Inlining and Devirtualization Are Prohibited

Condition	Prohibition Reason
Function address taken	Must preserve callable identity; prevent inlining that changes observable linkage
Interposable symbol	Loader may substitute implementation at runtime
Unresolved dynamic type	Cannot guarantee correct vtable dispatch elimination
Excessive code growth detected	Inliner cost model rejects to preserve I-cache locality
Volatile or atomic synchronization boundaries	Prevent transformations that reorder required semantics

These prohibitions ensure semantic correctness and prevent pathological performance outcomes.

5.4.7 Summary

Transformation	Input Condition	Output Form	Optimization Effect
Inlining	Body visible and not interposable	Call replaced by body copy	Expands optimization domain
Devirtualization	Dynamic type proven	Virtual call → direct call	Enables inlining and call elimination

Transformation	Input Condition	Output Form	Optimization Effect
No-Op (fallback)	Visibility or dynamic type unknown	Call preserved	Preserves correctness but limits optimization

Inlining and devirtualization are **semantic exposure mechanisms**.
They allow GIMPLE to express call targets explicitly, enabling propagation, folding, SSA simplification, alias refinement, and loop/vectorization pipelines.
Their correctness depends directly on **visibility, escape analysis, type inference, and profile data** results.

5.5 Examples: GIMPLE CFG Dissection with Dominator Tree Reconstruction

Control-Flow Graph (CFG) construction is performed after GIMPLE canonicalization and before SSA form construction. The CFG expresses basic blocks and edges representing possible execution paths. The **dominator tree** is derived from the CFG and is fundamental to optimization passes involving dead code elimination, loop detection, induction variable analysis, and value propagation.
This section presents a minimal example and reconstructs:

1. Basic block structure
2. CFG edge relationships
3. Dominator tree structure
4. Post-dominator relationships relevant to cleanup and exception edges

5.5.1 Example Source

```
int f(int x) {
    int r = 1;
    if (x > 0)
        r = x * 2;
    else
        r = 0;
    return r + 1;
}
```

Compile with IR dumps:

```
g++ -O0 -fdump-tree-cfg -c example.cpp
```

Relevant GIMPLE (simplified, comments added):

```
f (int x)
{
  int r;
  <bb 2>:
    r = 1;
    if (x > 0)
      goto <bb 3>;
    else
      goto <bb 4>;

  <bb 3>:
    r = x * 2;
    goto <bb 5>;

  <bb 4>:
    r = 0;

  <bb 5>:
    return r + 1;
}
```

5.5.2 CFG Block Structure

Block	Role	Successors
BB2	Entry and branch decision	BB3, BB4
BB3	True branch body	BB5
BB4	False branch body	BB5

Block	Role	Successors
BB5	Merge and function exit	Return

CFG Graph (textual):

```
      BB2
     /   \
   BB3   BB4
     \   /
      BB5
```

This is a canonical diamond-shaped conditional region.

5.5.3 Dominator Tree Construction

A block *D dominates B* if every path from the entry to B passes through D.
Dominance relationships:

Block	Immediate Dominator (IDOM)
BB2 (entry)	None
BB3	BB2
BB4	BB2
BB5	BB2

Dominator tree:

```
BB2
  BB3
  BB4
  BB5
```

Observations:

- BB2 dominates both branches and the merge.
- BB3 and BB4 do **not** dominate BB5 individually; BB5 is reachable through both.

5.5.4 Post-Dominator Relationships

A block *P post-dominates B* if every path from B to exit passes through P.
Post-dominators:

Block	Immediate Post-Dominator
BB2	BB5
BB3	BB5
BB4	BB5
BB5	None (exit)

Post-dominator tree:

```
BB5
  BB3
  BB4
  BB2
```

This ordering is used for:

- Tail merging
- Guarded expression hoisting
- Region failure cleanup paths

5.5.5 Dominance Relevance to Optimization

Correct dominator tree formation directly impacts:

Optimization Stage	Dominance Dependency
Dead Code Elimination	Identify unreachable blocks
Loop Recognition	Detect backedges where a block dominates its successor
Induction Variable Analysis	Identify loop header blocks
Scalar Replacement of Aggregates	Ensure uniqueness of reaching definitions
Constant Propagation	Requires dominance to prove stable reaching values

Example: If `r` remains unmodified along all dominated paths, `return r + 1` can be constant-folded once the variable stabilizes.

5.5.6 CFG and Dominator Diagnostics

Useful inspection flags:

```
-fdump-tree-cfg
-fdump-tree-dom
-fdump-tree-ssa
```

Interpretation:

- `cfg` → control-flow and basic block boundaries
- `dom` → dominator and post-dominator sets

- `ssa` → -function placement dependent on dominance frontier

Dominator frontier computation defines where -functions must be placed for SSA correctness.
In the example, insertion is **not required** because `r` is uniquely assigned per control region and merged explicitly in BB5.

5.5.7 Summary

Representation	Purpose	Output Structure
CFG	Encodes legal execution paths	Directed block graph
Dominator Tree	Defines structurally mandatory path prefixes	Parent-child dominance relationships
Post-Dominator Tree	Defines mandatory return-path convergence	Reverse direction tree formation
Dominance Frontier	Determines -node placement in SSA	Structural merge boundary set

CFG structural analysis and dominator tree derivation are **prerequisites** for SSA formation, loop normalization, scalar propagation, and high-level optimizations that rely on **execution path invariants**.

Part III

GIMPLE/SSA MIDEND AND OPTIMIZATION THEORY

Chapter 6

SSA Form Construction and Value Flow Algorithms

6.1 Phi-Node Insertion Rules and SSA Dominance Frontier

Static Single Assignment (SSA) form requires that every variable have a unique definition site. When control flow merges, multiple reaching definitions may converge at a block. **-functions** are inserted at merge points to unify these alternative definitions into a single SSA name. GCC constructs SSA form after CFG and dominator tree formation, using the **dominance frontier** to identify minimal -placement locations.

6.1.1 Reaching Definition Conflicts

Consider a variable `v` defined in multiple control-flow regions:

```
if (cond)
```

```
    v = a;
else
    v = b;
return v;
```

In SSA, both assignments must produce distinct names:

```
v1 = a;
v2 = b;
v3 =  (v1, v2);
return v3;
```

The -node models **control-dependent value selection**, not runtime branching logic.

6.1.2 Dominance Frontier Definition

For a basic block `B`, the **dominance frontier** `DF(B)` is the set of successor blocks `S` such that:

- `B` dominates a predecessor of `S`,
- but `B` does *not* strictly dominate `S`.

This indicates where execution paths merge after diverging at `B`. SSA -placement occurs at **dominance frontiers of variable definition blocks**.

6.1.3 Algorithm for Minimal -Node Insertion

For each variable `v`:

1. Collect all blocks `Def[v]` where definitions of `v` occur.
2. For each block `B` in `Def[v]`, traverse dominance frontier sets.

3. Insert -nodes in each block in `DF(B)` for `v`.

4. If -nodes introduce new definitions (new SSA names), iterate until fixpoint.

This ensures **minimal SSA form** (no redundant -functions).

6.1.4 Example Control Structure

Source:

```
int f(int x) {
    int r = 0;
    if (x > 0)
        r = x;
    else
        r = -x;
    return r;
}
```

CFG:

```
        BB2
       /   \
     BB3   BB4
       \   /
        BB5
```

Dominance Frontier:

- `DF(BB3) = {BB5}`
- `DF(BB4) = {BB5}`

Thus, -placement:

```
BB3: r1 = x;
BB4: r2 = -x;
BB5: r3 =  (r1, r2);
return r3;
```

BB5 does not dominate BB3 or BB4; therefore, the -node is needed.

6.1.5 SSA Name Binding and Use-Chain Maintenance

After -insertion:

- Each assignment to v now becomes a unique SSA name.
- Each use of v is replaced with the closest dominating SSA name based on CFG dominance.
- -functions introduce new SSA names that may themselves propagate to uses.

GCC maintains:

- **def-use chains** (DU chains)
- **use-def chains** (UD chains)

These allow constant propagation, dead code elimination, and alias reduction to operate on graph relationships rather than symbolic variable identifiers.

6.1.6 Cases Where Insertion Is Suppressed

 -nodes are not inserted when:

Case	Reason
Only one reaching definition	No selection needed
Reaching definitions are provably equivalent	Constant folding collapses ϕ
Variable stored in memory, not SSA-promotable	Memory SSA handles separately

Memory-resident objects require *load/store SSA*, not register SSA.

6.1.7 Summary

Concept	Definition	Purpose
Dominance Frontier	Boundary where control-flow paths reconverge	Determines minimal - placement
SSA -node	Merge of multiple reaching values	Ensures single assignment naming discipline
Def-Use Chains	Graph mapping between definitions and uses	Enables global propagation and elimination
Minimal SSA Form	SSA with no redundant - functions	Reduces IR noise and improves analysis efficiency

The construction of SSA using dominance frontier placement ensures a **structurally minimal and semantically correct value-flow representation**. This representation is the required substrate for all mid-end optimizations including constant propagation, loop induction inference, scalar replacement, and alias analysis.

6.2 Sparse Conditional Constant Propagation (SCCP)

Sparse Conditional Constant Propagation (SCCP) is a **hybrid propagation and pruning algorithm** that simultaneously performs constant propagation and dead code elimination in SSA form. Unlike classical dense propagation, SCCP limits analysis to the subset of the control-flow graph proven to be executable and the subset of SSA values proven to influence reachable computation. This yields a more precise and computationally efficient propagation model.
SCCP operates on **three parallel state lattices**:

1. **Value Lattice** — constant, non-constant, or undefined state for each SSA name.
2. **Execution Lattice** — reachable or unreachable state for each basic block.
3. **Edge Lattice** — feasible or infeasible state for control-flow edges.

This tri-lattice framework allows SCCP to remove unreachable control paths and fold run-time decisions into compile-time deterministically.

6.2.1 Value Lattice for SSA Names

Each SSA value `v` is assigned a state:

State	Meaning	Optimization Implication
Undefined	No known assignments yet	Candidate for constant propagation
Constant(c)	Expression evaluates to literal `c`	Replace uses with literal, fold operations

State	Meaning	Optimization Implication
Overdefined	Multiple non-equal values reach `v`	No constant folding possible

Transitions:

```
Undefined → Constant(c)
Constant(c) → Overdefined   (if conflicting assignment found)
```

This monotonic lattice guarantees convergence.

6.2.2 Control-Flow Feasibility Tracking

Rather than assuming all execution paths are possible, SCCP propagates **reachability**:

Block State	Meaning
Executable	Proven to be reachable
Not Executable	No feasible entry path established

Branch propagation uses SSA-known values:

```
if (cond) goto A else goto B
```

- If `cond` is Constant(true) → edge to A is **feasible**, edge to B is **infeasible**.
- If `cond` is Constant(false) → A is infeasible, B is feasible.
- If `cond` is Overdefined → both edges tentatively feasible.

This directly eliminates unreachable blocks.

6.2.3 SCCP over -Nodes

For -functions:

```
x3 = (x1 from BB3, x2 from BB4)
```

Only **values from executable incoming edges** contribute. If only one live predecessor remains, the collapses:

```
x3 = x1
```

If multiple constant inputs exist and differ → `x3` becomes Overdefined.

6.2.4 Instruction Folding Rules

SCCP folds any operation where operands are Constant:
Example GIMPLE:

```
t1 = 6
t2 = 7
t3 = t1 * t2   → folded to 42
```

After SCCP:

- `t3` is Constant(42).
- All uses of `t3` are replaced with literal `42`.
- Algebraic simplification is deferred to subsequent passes.

6.2.5 Elimination of Dead Branches and Blocks

Once block reachability converges:

- Blocks marked Not Executable are removed.
- Edges marked infeasible are pruned from CFG.
- Associated -arguments from eliminated edges are dropped.

Example:

```
if (0) { ... } else { ... }
```

transforms to:

```
goto else-block   (true branch removed entirely)
```

This reduces CFG complexity and supports subsequent loop and scalar optimizations.

6.2.6 Resulting IR Guarantees

After SCCP:

Property	Result
All foldable expressions replaced with constants	Strength reduction of arithmetic and comparisons
All unreachable control-flow removed	CFG simplification and reduced -placement
-functions minimized	Eliminated where merging is no longer needed

Property	Result
SSA uses now stabilized	Enables stronger global value numbering and alias pruning

SCCP thus forms a critical early optimization stage that **stabilizes the SSA graph** and maximizes later optimization potential.

6.2.7 Summary

Component	Purpose
Value Lattice	Tracks constant propagation through SSA values
Execution Lattice	Removes unreachable control paths
Simplification	Resolves merge nodes under reduced path sets
Folding and DCE	Produces simplified and smaller CFG and value graph

Sparse Conditional Constant Propagation is a **reachability-aware constant propagation algorithm** that directly improves SSA precision, reduces the size of the IR, and exposes optimization opportunities for loop inference, alias analysis, register promotion, and vectorization.

6.3 Range Propagation and Provenance Tracking

Range propagation is the process of assigning **value interval constraints** to SSA names based on control-flow conditions, arithmetic operations, and semantic context. Provenance tracking extends this by recording **origin relationships** between values, enabling elimination of contradictory conditions and refinement of data-flow ranges across the SSA graph. These analyses allow GCC to reason about numeric bounds, eliminate redundant checks, simplify branches, and improve vectorization safety.
GCC implements range reasoning through the **Ranger framework**, which performs **demand-driven, path-sensitive interval inference** over SSA form. The propagated constraints are integrated into conditional fold logic and loop analysis.

6.3.1 Value Range Lattice

Each SSA name `v` is assigned an interval:

```
Range(v) = [lower_bound, upper_bound]
```

Additionally, value ranges may be:

State	Meaning
Full	Unconstrained domain (no useful restriction)
Constant(c)	Singleton interval `[c, c]`
Semi-Bounded	One-sided interval, e.g., `[0, +∞)`
Empty	Discovered contradiction → block infeasible

This lattice is monotonic under refinement:

```
Full → Semi-Bounded → Constant → Empty
```

No expansion of ranges occurs once a bound has been established.

6.3.2 Sources of Range Information

Range constraints arise from:

1. **Comparisons in conditional branches**
   Example:

   ```
   if (x > 5) → Range(x) becomes [6, +∞) along true edge
   ```

2. **Arithmetic operations** with known operand bounds
   Example:

   ```
   y = x + 3 → Range(y) = Range(x) shifted by +3
   ```

3. **Standard library functions**, when recognized intrinsic
   Example:

   ```
   abs(x) → Range(abs(x)) = [0, max(|Range(x)|)]
   ```

4. **Loop induction variables** derived from canonical loop form

Ranger evaluates range queries lazily, expanding dependencies only when needed.

6.3.3 Branch-Sensitive Propagation

Range constraints are **path-specific**:

```
if (x < 10)
    ...
else
    ...
```

CFG is updated with edge conditions:

Edge	Propagated Range
True branch	`Range(x)  (-∞, 9]`
False branch	`Range(x)  [10, +∞)`

These refined ranges propagate forward through SSA use-def chains.
Infeasible branches occur if a condition contradicts existing ranges.
When detected, the block is marked unreachable and removed in cleanup passes.

6.3.4 Provenance Tracking

Provenance records **origin dependencies** between SSA values:

```
v2 originates from v1 via operation 0
```

Tracking provenance enables:

- Reverse flow reasoning for alias classification
- Elimination of redundant comparisons (e.g., `x > 5` repeated after its guard)
- Correlation inference across differently derived SSA names

Example:

```
y = x + 1
if (y > 10) → implies x > 9
```

This protects propagation from losing precision when values are represented indirectly.

6.3.5 Loop-Carried Range Refinement

Loop analysis establishes induction variable patterns:

```
for (i = 0; i < n; i++)
```

Ranger derives:

```
Range(i) = [0, n-1] across the loop body
```

This allows:

- Loop bound tightening
- Vectorization legality checks
- Removal of redundant array bounds checks (when proven safe)

Range inference across loop iterations is performed using **widening**, ensuring guaranteed termination.

6.3.6 Integration with Optimization Stages

Range and provenance data directly feed:

Optimization Stage	Use
Conditional Constant Propagation (CCP/SCCP)	Refined branch pruning
Dead Code Elimination	Removal of infeasible blocks
Store/Load Reassociation	Proof of non-overlapping access regions

Optimization Stage	Use
Loop Strength Reduction	Safe rewrite of induction variable steps
Vectorization Planning	Alignment and bounds safety inference

The correctness of vectorization often depends on successful range proof.

6.3.7 Summary

Component	Purpose	Result
Range Propagation	Compute numeric bounds for SSA values	Determines feasible execution conditions
Provenance Tracking	Track derivation relationships	Enables reverse inference and condition elimination
Path-Sensitive Refinement	Assign ranges per control-flow edge	Eliminates infeasible branches
Loop Range Analysis	Infer induction variable intervals	Enables safe and profitable loop optimizations

Range propagation and provenance tracking provide GCC with a **numerically constrained SSA graph**, establishing provable correctness boundaries necessary for branch elimination, vectorization, and alias reduction.

6.4 Escape, Escape-Not-Escape, and Escape Set Inference

Escape analysis in GCC determines whether an object, pointer, or reference value may become visible outside its defining scope or function. This determines eligibility for stack allocation, scalar replacement, and alias pruning. The inference engine builds **escape sets** describing which memory regions or SSA names are reachable from external or unknown contexts.
An SSA name or memory object is said to *escape* if:

1. Its address is stored into memory accessible by another function.
2. It is returned by the current function.
3. It is passed as a pointer or reference to a call with unknown side effects.
4. It is captured in a closure or stored into global/static storage.

Escape inference is executed after early GIMPLE canonicalization and before alias classification to minimize unnecessary heap or global assumptions.

6.4.1 Object and Reference Escape Classification

Each SSA name or object allocation is classified into one of three categories:

Category	Definition	Optimization Eligibility
Escape	Object potentially visible outside defining context	Must assume global alias; cannot scalarize

Category	Definition	Optimization Eligibility
Escape-Not-Escape (Partial Escape)	Escapes conditionally or through limited alias	May remain stack-allocated under guarded constraints
Non-Escape	Object provably local to function or lexical block	Eligible for scalar replacement and stack promotion

Partial escape analysis refines conservative results where address-taken conditions depend on control flow or constrained parameter passing.

6.4.2 Escape Source Identification

Escape sources are detected by pattern matching over GIMPLE instructions and SSA use-def chains. Typical triggers:

- `ADDR_EXPR` or `&object` stored in memory or assigned to global.
- `MEM_REF` to an address escaping through a function call argument.
- Calls marked with unknown or non-pure side effects.
- Assignments across function boundaries.

Example (simplified GIMPLE):

```
p_1 = &x;
call foo(p_1);       // Escape through parameter
q_2 = &y;
use(q_2);            // Non-escape if 'use' is local and pure
```

GCC annotates `x` as escaping, `y` as non-escaping.

6.4.3 Escape Set Construction

An **escape set** is a data-flow abstraction defining which objects or SSA names may reference escaping storage.
Algorithmic steps:

1. Initialize `EscapeSet` with all global and heap-allocated objects.
2. Propagate escape relationships:
    - If `p` points to an escaping object, mark all aliases of `p` as escaping.
    - If `q = p`, propagate escape state from `p` to `q`.
3. Iterate until fixed point (no new members added).

The resulting escape lattice is monotonic:

```
Non-Escape → Escape-Not-Escape → Escape
```

This ensures convergence under iterative data-flow analysis.

6.4.4 Escape-Not-Escape Refinement

Certain cases require intermediate state assignment:

- **Temporaries returned by value** but not by reference.
- **Captures in lambdas** where closure is non-escaping.
- **Pointers to stack-local arrays** used only in inline-expanded callees.

Example:

```
int* f() {
    int x;
    return &x;  // Escape
}

int g() {
    int y;
    return y + 1; // Non-escape
}
```

In GCC IR, x's address is marked escaping; y's address never materializes.
Conditional escapes are recorded with edge predicates in the control-flow graph.
Only the paths that satisfy the escape condition mark propagation into the escape set.

6.4.5 Relationship with Alias and Memory SSA

Escape information is a precondition for **Memory SSA** construction.
Memory SSA groups load/store operations into equivalence classes based on memory region identity.
Escape analysis ensures that only truly global or escaping objects participate in alias relationships.

Memory Operation	Escape Status	Optimization Impact
Load from non-escaping object	Non-aliasing; promotable	Can be replaced by scalar register value
Store to escaping object	Global side effect	Cannot eliminate or reorder
Load/store from partial-escape region	Guarded alias	Requires conditional dependence tracking

This classification improves precision of **store motion**, **dead store elimination**, and **pure-call optimization**.

6.4.6 Practical Outcomes of Escape Inference

Escape inference affects several compiler stages:

Stage	Effect
Stack Allocation	Objects marked non-escaping promoted from heap to stack
Scalar Replacement	Structs or arrays replaced with independent SSA scalars
Interprocedural Constant Propagation	Values of non-escaping pointers propagated interprocedurally
Code Motion	Movements restricted for escaping memory references
Parallelization	Escape-free loops eligible for private variable promotion

The correctness of escape inference directly influences register pressure and memory bandwidth utilization in optimized code.

6.4.7 Summary

Concept	Definition	Impact
Escape	Value or object reachable from external scope	Disables alias collapsing
Escape-Not-Escape	Partially escaping under path or call conditions	Conditional optimization permitted
Escape Set	Collection of values sharing external visibility	Drives alias and Memory SSA construction

Escape and non-escape inference provide the compiler with **semantic visibility boundaries**.

Accurate escape classification is essential for safe scalarization, precise alias modeling, and high-level optimizations in the GCC mid-end.

6.5 Examples: SSA Rewrites Under Aggressive Inlining Constraints

When inlining is applied aggressively, SSA form must be reconstructed to maintain single-definition semantics across expanded control-flow regions. Inlining replaces a call site with the body of the callee, introducing new variables, merge points, and potential alias interactions. GCC resolves this by performing **SSA renaming**, -node re-evaluation, and value propagation across the combined call graph fragment.
This section illustrates the resulting transformations using a minimal example that exposes both scalar propagation and -node normalization under inlining.

6.5.1 Example Source

```
static inline int g(int v) {
    if (v > 5)
        return v + 1;
    return v - 1;
}

int f(int x) {
    int a = g(x);
    return a * 2;
}
```

Compile at `-O3 -fdump-tree-ssa`:

```
g++ -O3 -fdump-tree-ssa -c example.cpp
```

6.5.2 Inlining Transformation Result (Conceptual GIMPLE Before SSA Fixup)

Without SSA renaming:

```
f(int x)
{
  int a;
  if (x > 5)
    a = x + 1;
  else
    a = x - 1;
  return a * 2;
}
```

This introduces multiple reaching definitions for `a`.
SSA rewriting must disambiguate them.

6.5.3 SSA Rewrite with -Node Placement

After SSA renaming:

```
f(int x_1)
{
  if (x_1 > 5)
    a_2 = x_1 + 1;
  else
    a_3 = x_1 - 1;

  a_4 = (a_2, a_3);
  t_5 = a_4 * 2;
  return t_5;
}
```

Key observations:

- The inline expansion **did not require** separate handling for `g`—it becomes a control-flow region inside `f`.
- The **dominance frontier** places `(a_2, a_3)` at the merge block.
- Multiplication sees a **single canonical SSA value** `a_4`.

6.5.4 Value Propagation and Constant Folding Interaction

If the call site provides additional semantic context, propagation may eliminate branch structure.
Example modified source:

```
int h() {
    return f(10);
}
```

Inlining of both `h()` and `f()`:

```
x_1 = 10;
if (10 > 5)        // condition is constant true
  a_2 = 11;
else
  (dead)
a_4 = a_2;
t_5 = a_4 * 2;     // 11 * 2 = 22
return 22;
```

Further SCCP and DCE remove the unreachable branch and -node:

```
return 22;
```

This demonstrates **multi-phase cross-function constant propagation** enabled by SSA merging.

6.5.5 Interaction with Escape and Alias Constraints

If a captured or referenced object is introduced during inlining, SSA must differentiate storage identity:
Example:

```
static inline void k(int& r) { r = r + 1; }
int m(int y) {
    k(y);
    return y;
}
```

After inlining:

```
y_2 = y_1 + 1;
return y_2;
```

y remains a **single SSA variable** because its storage does not escape; no -node is required.
If k had stored &r globally, the variable would instead be classified as **escaping**, preventing scalarization and forcing memory-based SSA.

6.5.6 Loop-Carried SSA Transformation Under Inlining

Inlining into a loop may introduce new induction variables requiring canonical restructuring:

```
for (i = 0; i < n; ++i)
    s += g(i);
```

After inlining and SSA:

```
i_1 = 0;
s_1 = 0;
loop:
  if (i_1 > 5) tmp_1 = i_1 + 1; else tmp_1 = i_1 - 1;
  s_2 = s_1 + tmp_1;
  i_2 = i_1 + 1;
  if (i_2 < n) goto loop;
return s_2;
```

SSA induction recognition may rewrite `(i_1, i_2)` into canonical loop-carried form. This establishes the basis for loop vectorization and strength reduction.

6.5.7 Summary

Optimization Interaction	**SSA Effect**	**Resulting Opportunity**
Inlining	Introduces new SSA name regions	-node placement and renaming required
Constant propagation	Simplifies branch structure	Dead branch and elimination
Escape analysis	Determines scalarization viability	Affects memory SSA region count
Loop integration	Normalizes induction variables	Enables vectorization and strength reduction

Aggressive inlining expands visibility of value flow, and SSA rewriting ensures that the merged region remains **structurally analyzable**, enabling the compiler to perform higher-order transformations correctly and efficiently.

Chapter 7

Control Flow Optimization, Loop Analysis, and Polyhedral Modeling

7.1 Loop Induction Variable Classification

Induction variable (IV) classification is the process of identifying variables whose values evolve linearly across loop iterations. GCC performs IV analysis on GIMPLE SSA form to enable loop normalization, strength reduction, dependence testing, vectorization, unrolling, and cost modeling. The classification is based on data-flow relationships and dominance structure, not syntactic loop syntax.
An induction variable is defined as an SSA name `v_k` that satisfies:

```
v_{k+1} = v_k   C
```

where is an associative arithmetic operator (typically addition or subtraction) and `C` is a loop-invariant constant. The induction update must be dominated by the loop header and form a **cycle** in SSA use-def relations.

7.1.1 Detection of Basic Induction Variables (BIVs)

A **basic induction variable** is introduced at loop entry and incremented once per iteration along the loop backedge.
Canonical detection pattern in GIMPLE SSA:

```
Loop Header:
   i_1 = PHI(i_entry, i_2)
Loop Body:
   i_2 = i_1 + C   (C is loop-invariant)
Backedge:
   goto Loop Header
```

Conditions:

- The -function defining `i_1` must have one incoming value from outside the loop and one from inside.
- The increment expression must be dominated by the loop latch.
- The loop-carried dependency must be unique.

BIVs define loop iteration count and enable structural loop reasoning.

7.1.2 Derived Induction Variables (DIVs)

A **derived induction variable** is a function of a BIV and loop-invariant parameters:

```
d = a * i + b
```

where `a` and `b` are loop-invariant.
DIV detection is performed using symbolic substitution on SSA expressions. Ranger's interval propagation further bounds DIV ranges to support alias disambiguation and array bounds reasoning.

This transformation allows index expressions to be simplified even when expressed indirectly.

7.1.3 Invariants vs. Induction Variables

A variable `v` is considered **loop-invariant** if:

- All definitions of `v` occur outside the loop, or
- All operands of expressions computing `v` resolve to loop-invariant SSA names.

Invariant recognition is critical because:

Classification	Optimization Result
Loop-invariant	Hoist outside loop (LICM)
Basic induction	Normalize to canonical iteration form
Derived induction	Expand to affine index expressions for vectorizer

This separation establishes a strict algebraic foundation for loop optimization.

7.1.4 Induction Variable Normalization

To support vectorization and polyhedral transformation, GCC rewrites induction updates into canonical increment-by-constant form:

```
i_next = i + 1
```

If the original loop step differs, scalar evolution analysis rewrites the expression accordingly:

```
i_next = i + stride
```

Normalization requires that stride is provably loop-invariant.
If normalization fails, high-level vectorization opportunities are reduced or disabled.

7.1.5 Induction Variables in Nested Loops

For nested loops, IV classification is hierarchical:

```
Outer Loop: i
Inner Loop: j
```

Independence conditions:

- `j` is an IV relative to the inner loop only.
- `i` is invariant relative to the inner loop.
- Combined affine expressions of the form `A*i + B*j + C` are processed by the scalar evolution (SCEV) engine.

Correct hierarchical IV classification is required for legality checks in:

- Loop interchange
- Strip-mining
- Fusion and fission scheduling
- Polyhedral model extraction

7.1.6 Relation to Dependence Testing and Vectorization

Induction variable classification feeds into dependence tests:

- Scalar evolution provides closed-form expressions of access patterns.
- Bounds analysis uses IV ranges to verify memory safety.
- Vectorizer uses normalized IVs to map iteration space to vector lanes.

If induction classification proves:

```
A[i] and A[i + k] do not overlap
```

then vectorization is enabled under memory independence guarantees.

7.1.7 Summary

Component	Role	Optimization Impact
Basic Induction Variable (BIV)	Defines iteration count and loop progression	Enables normalization and scalar evolution
Derived Induction Variable (DIV)	Index or offset expression derived from BIV	Allows array indexing simplification
Loop-Invariant Value	Does not depend on iteration state	Enables code motion and hoisting
Scalar Evolution (SCEV)	Computes closed-form iteration expressions	Required for vectorization and polyhedral analysis

Induction variable classification provides the **algebraic foundation** for all loop-centric optimizations.
Without precise IV structure inference, the mid-end cannot legally transform loops for speed, memory efficiency, or parallel execution.

7.2 Loop Invariant Code Motion and Peeling vs Unrolling

Loop optimization in GCC distinguishes computations that depend on loop iteration state from computations that do not. **Loop Invariant Code Motion (LICM)** moves invariant computations outside the loop to reduce the dynamic instruction count. **Loop peeling** and **loop unrolling** restructure loop iteration boundaries to enable further optimizations such as vectorization, constant propagation, and induction variable simplification. Selection of these transformations depends on cost models and legality conditions derived from SSA form and alias analysis.

7.2.1 Loop Invariance Detection

A value `v` is considered **loop-invariant** if:

1. All SSA definitions reaching `v` occur outside the loop, or
2. All operands in the computation of `v` are loop-invariant and side-effect free.

GCC identifies invariants using:

- Dominator analysis on SSA definitions,
- Ranger-based range and bounds inference,
- Memory SSA alias classification to ensure non-interference on referenced storage.

Example (conceptual transformation):
Source:

```
for (int i = 0; i < n; ++i)
    y += a * x[i];
```

Lowered GIMPLE (before LICM):

```
t1 = a;          // a is invariant
loop:
  t2 = x[i];
  t3 = t1 * t2;   // multiplication repeated every iteration
  y += t3;
```

After LICM:

```
t1 = a;            // moved before loop
loop:
  t2 = x[i];
  y += t1 * t2;
```

If `a` is a compile-time constant, constant propagation may fold its uses further.

7.2.2 Correctness Requirements for LICM

To move an instruction out of a loop:

Condition	Requirement
Memory safety	The operation does not access loop-variant memory locations
No side effects	The instruction must not participate in I/O, volatile memory access, or synchronization
Single reaching definition	All operands must be invariant under loop iteration

If alias classification cannot separate memory regions, LICM is conservatively disabled.

7.2.3 Loop Peeling

Peeling executes one or more iterations of the loop header before entering the main loop body.
Peeling is applied to:

- Simplify conditional checks that are iteration-dependent,
- Eliminate induction variable edge cases,
- Align memory accesses for vectorization.

Example case enabling vector alignment:

```
while (i < n && (uintptr_t)&x[i] % 32 != 0)
    peel iteration;
```

Once alignment constraints are achieved, the main loop body becomes vectorizable. Peeling does not change iteration count; it adjusts entry boundary semantics.

7.2.4 Loop Unrolling

Unrolling replicates the loop body multiple times per iteration to reduce control overhead and expose ILP (Instruction-Level Parallelism):
Example unrolling by 4:

```
for (i = 0; i < n; i += 4) {
    body(i);
    body(i+1);
    body(i+2);
    body(i+3);
}
```

Benefits:

- Fewer branch evaluations.
- Increased availability of independent operations for scheduling.
- Improved pipeline utilization.

Costs:

- Increased code size.
- Higher register pressure, potentially causing spill code.

GCC applies unrolling selectively based on block hotness (profile data) and available registers (target machine model).

7.2.5 Peeling vs. Unrolling: Distinct Goals

Transformati	Purpose	Driven By	Typical Impact
Peeling	Restructure entry/exit conditions	Alignment, boundary checks	Enables vectorization and simplifies flow
Unrolling	Replicate work within loop	ILP and branch reduction	Improves throughput at cost of code size

Peeling changes **initial execution behavior** to regularize structure.
Unrolling changes **loop granularity** to exploit hardware parallelism.
They are often applied in sequence: peel → vectorize → unroll.

7.2.6 Interaction with Scalar Evolution (SCEV)

Scalar Evolution provides closed-form recurrence expressions for induction variables, which allows:

- Detecting that invariants are safe to hoist.
- Determining peel counts to eliminate conditional guards.
- Choosing unroll factors that preserve correctness.

Example:
If loop bounds are known:

```
for (i = 0; i < n; ++i)
```

and alignment requires:

```
(p + i) mod 8 == 0
```

Peel count `k = ((-p) mod 8)` follows directly from the SCEV representation.

7.2.7 Summary

Component	Role	Optimization Enabled
Loop Invariant Code Motion	Remove redundant intra-loop computation	Reduces dynamic execution cost
Loop Peeling	Normalize loop-entry conditions	Enables vectorization and simplifies iteration bounds

Component	Role	Optimization Enabled
Loop Unrolling	Expand operations per iteration	Increases ILP and reduces branch overhead
Scalar Evolution	Provides recurrence proof and index modeling	Guides legality and cost decisions

LICM, peeling, and unrolling form the **transformation foundation** for modern loop optimization.
Their correctness and profitability depend on SSA value analysis, alias classification, and iteration space modeling.

7.3 Alias Analysis and Dependence Graph Construction

Alias analysis determines whether different memory references may refer to the same storage location. Dependence graph construction extends this by modeling ordering constraints between memory accesses within and across loop iterations. These analyses are prerequisites for loop transformations including interchange, fusion, fission, vectorization, and parallelization. In GCC, alias reasoning integrates **Memory SSA**, **points-to propagation**, and **range/provenance inference** to produce a unified dependence model.

7.3.1 Memory Reference Classification in GIMPLE

Each memory reference is categorized according to its **storage identity**:

Reference Type	Typical Examples	Alias Behavior
Local scalar promoted to SSA	Stack or temporary values	Guaranteed non-aliasing; no memory reference remains
Object-specific memory	Local arrays, closure fields, `new` allocations	Alias restricted to originating region
Global or externally visible storage	Globals, static variables, captured pointers	Conservative alias assumptions
Unknown pointer-derived access	Indirect loads/stores via pointers	Requires points-to and escape analysis

Only references remaining in memory space participate in alias modeling.

7.3.2 Points-to Set Inference

Each pointer-typed SSA name `p` is associated with a **points-to set** representing possible pointee locations:

```
Pts(p) = {Memory Regions R1, R2, ...}
```

Inference steps:

1. **Initialization:**
    - Direct address expressions form singleton sets (`&x → {x}`).
2. **Propagation through assignments:**
    - `q = p` implies `Pts(q)  Pts(p)`.
3. **Propagation through loads/stores:**
    - `*p = ...` associates region-level side effects to each region in `Pts(p)`.
4. **Constraint-based narrowing:**
    - If loops or branches restrict pointer evolution, Ranger supplies tightened value bounds.

When `Pts(p)` contains a single region, alias relations simplify substantially.

7.3.3 Memory SSA Region Graph

Memory SSA represents memory state changes as **distinct versioned SSA names**:

```
mem_1
mem_2 = store(mem_1, x)
y = load(mem_2)
```

Memory -functions are inserted at control-flow merge points:

```
mem_4 =  (mem_2, mem_3)
```

This produces a **def-use chain for memory**, enabling:

- **Dead store elimination** when stores do not influence surviving loads.
- **Hoisting and sinking** of memory operations when proven safe.
- **Loop dependence detection** because memory flow is explicit.

7.3.4 Dependence Classification in Loops

A memory dependence exists between two accesses `A` and `B` if:

1. They reference overlapping memory regions, and
2. At least one of them writes.

Dependence types:

Type	Meaning	Effect on Transformation
Flow (true)	Write → Read ordering	Constraints loop-carried parallelism

Type	Meaning	Effect on Transformation
Anti	Read → Write ordering	Potential reordering only with care
Output	Write → Write ordering	Must preserve write ordering
Input	Read → Read	No effect on reordering

If ranges prove non-overlap (e.g., disjoint array segments), dependence collapses to **Input**, enabling vectorization and parallel execution.

7.3.5 Dependence Graph Construction

The loop dependence graph (LDG) is constructed as follows:

1. Enumerate memory references within the loop body.
2. Determine alias compatibility using points-to results and region-level classification.
3. Determine access ordering using SSA dominance and loop-carried -values.
4. Insert directed edges representing dependence type and strength.

Edges annotated with stride and range bounds allow the vectorizer and polyhedral engine to infer affine access patterns.

7.3.6 Application to Loop Interchange, Fusion, and Vectorization

Transformation	Dependence Requirement	Enforcement Mechanism
Interchange	No carried dependences across loop levels	LDG must remain acyclic under level swap
Fusion	Accesses must be compatible in iteration order	LDG must not introduce new carried dependences
Vectorization	No loop-carried true dependencies	LDG edges must be proven non-carried via range/SCEV analysis

Failure to satisfy dependence constraints **disables** the transformation; GCC does not speculate correctness.

7.3.7 Summary

Component	Purpose	Output
Points-to Analysis	Determine possible referent objects of pointers	Memory region identity sets
Memory SSA	Represent memory state evolution explicitly	Versioned memory values and -nodes
Dependence Graph	Represent ordering constraints among accesses	Directed graph informing transformation legality
Scalar Evolution + Range Analysis	Prove disjoint access intervals	Enables safe reordering and parallel execution

Alias analysis and dependence graph construction provide the **semantic safety**

guarantees required for loop restructuring, vectorization, and high-performance code generation. The accuracy of these analyses directly determines whether the optimizer can legally and profitably transform computational kernels.

7.4 Introduction to Graphite / isl Polyhedral Optimizer

The **Graphite** framework in GCC performs high-level loop and data-flow transformations using the **polyhedral model**, with the **isl (Integer Set Library)** providing symbolic set and relation manipulation. Graphite operates on loop nests represented in GIMPLE SSA form after normalization, extracting their iteration spaces and memory access functions into an abstract representation suitable for dependence testing, restructuring, and scheduling optimization. Polyhedral transformations allow systematic and provably correct reordering of iteration spaces, enabling locality improvement, parallel execution, and vectorization alignment.

7.4.1 Polyhedral Representation Model

A loop nest is represented as:

1. **Iteration Domain** `D`
   The set of all integer tuples `(i , i , …, i )` describing legal loop iterations.

2. **Access Relations** `A`
   Mappings from iteration tuples to memory locations.

3. **Scheduling Function** `S`
   A mapping assigning execution timestamps or orderings to iteration space points.

For a loop of form:

```
for (i = L1; i < U1; i++)
  for (j = L2; j < U2; j++)
    S[i][j] = ...
```

The iteration domain:

```
D = { (i, j)   ² | L1   i < U1   L2   j < U2 }
```

Access patterns, such as `A[i][j] = X[i][j+1]`, are represented as affine maps.

7.4.2 Extraction from GIMPLE to Polyhedral IR

Graphite requires:

- Canonical loop form (single index, affine bounds).
- Loop-invariant increment stride.
- Absence of irreducible control flow in the loop region.
- Memory references expressible as affine functions of loop indices.

These constraints are checked by the **SCEV and range analysis subsystem** before polyhedral extraction. Non-conforming loops bypass Graphite.

7.4.3 Dependence Testing and Legality

Graphite constructs dependence relations `R` from access relations:

```
R = { (x, y) | x precedes y   A(x) = A(y)   at least one access is a write }
```

Legality condition:

```
S(new) must preserve the partial order induced by R.
```

This guarantees **semantic equivalence** of the transformed loop schedule.
isl performs dependence discovery and schedule search using integer set constraint solving.
If dependence constraints are violated, the transformation is rejected, not approximated.

7.4.4 Transformation Classes Performed by Graphite

Transformation	Objective	Example Effect
Loop Interchange	Improve stride-1 locality	Swap nesting of `i` and `j` loops
Loop Tiling (Blocking)	Improve cache reuse	Split iteration space into rectangular tiles
Strip Mining	Prepare for vectorization or GPU mapping	Convert a loop into chunks of fixed size
Loop Fusion / Fission	Reduce overhead or increase parallel granularity	Merge or separate adjacent loop nests
Affine Scheduling	Reorder entire iteration lattice	Minimize distance across dependence edges

All transformations are **global to the loop nest**, not performed heuristically per-statement.

7.4.5 Integration with the Mid-End Optimization Pipeline

Graphite operates between SSA canonical optimization and late loop/vector optimizers: Pipeline position (simplified):

```
GIMPLE SSA → Loop Canonicalization → Scalar Evolution → Graphite (polyhedral) →
↪  Vectorizer / Unroller → RTL Lowering
```

After Graphite emits a transformed SCoP (Static Control Part), loops are reconstructed back into GIMPLE with modified loop structures and access patterns.

7.4.6 Practical Constraints in Real-World Codebases

Graphite is effective when:

- Loops have regular affine bounds.
- Memory access functions depend linearly on index variables.

Graphite is less effective or bypassed when:

- Complex pointer arithmetic prevents affine recognition.
- Data-dependent control flow exists inside the loop.
- Memory access patterns require non-affine indexing.

In practice, **scientific kernels, DSP pipelines, and tensor algebra** benefit most from polyhedral restructuring.

7.4.7 Summary

Component	Role	Output
isl Integer Set Library	Solves affine constraints and dependence systems	Legal iteration schedules
Graphite Extractor	Converts GIMPLE loops to polyhedral representation	Iteration domains and access relations
Schedule Optimizer	Searches for improved execution ordering	Transformed loop nests respecting dependencies

Component	Role	Output
Loop Rebuilder	Re-generates GIMPLE from transformed schedule	Optimized control-flow ready for vectorization

The polyhedral optimizer provides the compiler with the ability to apply **mathematically proven-correct loop restructuring**, enabling large, global performance shifts in well-structured numerical code.

7.5 Examples: Loop Vectorization Feasibility Prediction Diagnostics

Vectorization feasibility in GCC is determined by a sequence of legality and profitability checks. These checks analyze memory access patterns, induction structure, aliasing guarantees, scalar evolution constraints, and control-flow uniformity. When vectorization is disabled, GCC emits diagnostic reasoning when enabled with `-fopt-info-vec` or `-fopt-info-vec-missed`.
This section demonstrates representative feasibility outcomes and the corresponding GIMPLE/SSA reasoning steps.

7.5.1 Example Loop

```
void f(float* __restrict x, float* __restrict y, int n) {
    for (int i = 0; i < n; ++i)
        y[i] = 3.0f * x[i];
}
```

Compile:

```
g++ -O3 -fopt-info-vec -c example.cpp
```

Typical diagnostic:

```
example.cpp:3: note: loop vectorized
```

Reason:

- Affine memory accesses (`x[i]`, `y[i]`)
- Basic induction variable recognized (`i`)

- No loop-carried dependencies
- Multiplication is vectorizable for target ISA (SSE/AVX)
- Pointer arguments marked with `__restrict` prevent alias uncertainty

The vectorizer confirms legality before profitability analysis.

7.5.2 Dependence-Inhibited Case

```
void g(float* a, float* b, int n) {
    for (int i = 0; i < n; ++i)
        a[i] = a[i] + b[i];
}
```

With no restrict qualifiers:

```
g++ -O3 -fopt-info-vec-missed -c example.cpp
```

Diagnostic:

```
example.cpp:4: note: not vectorized: potential aliasing prevents memory disambiguation
```

Internal reasoning:
`Pts(a)` and `Pts(b)` may overlap → alias classification cannot prove independence. Dependence graph includes possible **flow dependence**, inhibiting vectorization. Marking pointers `__restrict` or using `-fno-semantic-interposition` or LTO often resolves this.

7.5.3 Non-Affine Access Inhibition

```
void h(float* x, int* idx, int n) {
    for (int i = 0; i < n; ++i)
        x[i] *= x[idx[i]];
}
```

Diagnostic:

```
note: not vectorized: non-affine memory index pattern
```

Reason:
Memory access `x[idx[i]]` is indexed by a non-affine function of induction `i`. Dependence analysis cannot compute guaranteed independence → unsafe to vectorize. Graphite/isl cannot extract affine access relations → loop excluded from vector pipeline.

7.5.4 Masked Vectorization Consideration (Post-GCC 11)

Certain non-uniform control flows may still be vectorizable via **masking**, if hardware ISA supports predication (AVX-512):

```
void k(float* x, int n) {
    for (int i = 0; i < n; ++i)
        if (x[i] > 0)
            x[i] = -x[i];
}
```

Diagnostics:

```
note: loop vectorized using masked operations
```

Condition:

- The branch is uniform per element and does not introduce loop-carried dependence.
- Target architecture supports masked execution (AVX-512 or SVE).
- Profitability model indicates acceptable masked execution cost.

If mask cost > predicted throughput benefit → vectorization is declined.

7.5.5 Failures Due to Floating-Point Semantics

Example:

```
void m(float* x, int n) {
    for (int i = 0; i < n; ++i)
        x[i] = x[i] / (x[i] - 1.0f);
}
```

Diagnostic (default):

```
note: not vectorized: floating-point semantics prevent reassociation
```

Reason:

- Vectorization may change rounding order or exception signaling.
- Requires explicit permission for relaxed IEEE semantics:

```
g++ -O3 -ffast-math -fopt-info-vec
```

Enables reassociation and vectorization if allowed by user semantics.

7.5.6 Summary

Inhibition Cause	Vectorizer Reason	Possible Resolution
Alias uncertainty	Unsafe memory overlap	Add `restrict`, enable LTO, refine alias analysis
Non-affine indexing	Cannot construct affine access functions	Rewrite access, apply data-layout transformation
Control divergence	Branch divergence blocks SIMD	Masked vectorization if ISA supports it
FP semantic precision	Strict IEEE ordering prevents reassociation	Use `-ffast-math` or targeted pragmas
Unrecognized induction form	Loop cannot be normalized	Normalize loop or transform structure

Vectorization feasibility diagnostics are **evidence-based**, derived from SSA and dependence graphs.
The compiler does not guess or speculate correctness; transformations are enabled only when **proven safe and profitable**.

Part IV

RTL BACKEND AND TARGET MICROARCHITECTURE

Chapter 8

RTL Instruction IR and Machine Description Language

8.1 RTL Expression Trees and Operand Constraints

The **Register Transfer Language (RTL)** is GCC's low-level intermediate representation used after GIMPLE lowering and before instruction selection and register allocation. RTL models computation in a form that resembles abstract assembly, where each expression represents a transformation of machine-level operands. RTL is both **data-flow explicit** (operands and results are visible) and **target-sensitive**, since it interfaces directly with the machine description (MD) language that defines instruction encodings and operand legality.

8.1.1 RTL Expression Structure

Each RTL node is an S-expression encoded as a typed operator with operands. A generic RTL form:

```
(code operand  operand  ... operand )
```

Example:

```
(set (reg:SI 5) (plus:SI (reg:SI 5) (const_int 4)))
```

Meaning:

- Assign to register 5 (32-bit `SI` mode)
- The result of adding register 5 and the constant 4.

Key components:

Component	Meaning
code	Operation (e.g., `set`, `plus`, `mult`, `compare`)
mode	Data width and type (e.g., `QI`, `HI`, `SI`, `DI`, `SF`, `DF`)
operands	Registers, memory references, immediates, or subexpressions

RTL is strictly typed with respect to machine mode. Mismatched modes invalidate patterns during instruction matching.

8.1.2 Operand Categories

Operands in RTL expressions fall into distinct classes:

Operand Type	Example Form	Description
Register	`(reg:SI 5)`	Virtual or physical register

Operand Type	Example Form	Description
Memory reference	`(mem:SI (reg:DI 2))`	Indirection through address registers
Immediate constant	`(const_int 7)`	Literal integer encodable at instruction level
Address constant	`(symbol_ref:DI "label")`	Pointer to symbol or global
Complex addressing mode	`(plus:DI (reg:DI 1) (const_int 32))`	Address computation represented as RTL expression

These operand classes map directly to addressing and operand rules in the architecture's ABI.

8.1.3 Constraint Language for Instruction Operands

Machine descriptions specify instructions using **operand constraints**, which control what kinds of operands are legal for each operand position. Constraints enforce:

- Register class eligibility
- Immediate encoding restrictions
- Addressing mode capabilities
- Required reloads before allocation

Common constraint classes on x86-64:

Constraint	Meaning
`"r"`	Any general-purpose register
`"m"`	Memory operand allowed
`"i"`	Integer immediate that fits encoding limits
`"o"`	Memory operand with direct addressing only
`"A"`	Accumulator register operand (architecture-specific)

For example, an MD instruction pattern may specify:

```
(define_insn "addsi3"
  [(set (match_operand:SI 0 "register_operand" "=r")
        (plus:SI (match_operand:SI 1 "register_operand" "r")
                 (match_operand:SI 2 "immediate_operand" "i")))]
  ...
)
```

This enforces:

- Operand 0 must be assigned a register (`=r` means writable).
- Operand 1 must be a register.
- Operand 2 must be an immediate that satisfies machine encoding.

Incorrect constraints prevent valid match patterns during code generation.

8.1.4 RTL and Machine Modes

Modes encode **both width and semantics**:

Mode	Meaning
`SI`	32-bit integer
`DI`	64-bit integer
`SF`	32-bit IEEE float
`DF`	64-bit IEEE float
`V4SI`	Vector of 4× 32-bit integers

The backend uses modes to:

- Select appropriate instructions (`addsi3` vs `adddi3`)
- Enforce SIMD width alignment for vectorization
- Control register allocation to correct register classes (e.g., integer vs SIMD).

Incorrect mode inference stalls instruction selection or forces unnecessary data movement.

8.1.5 RTL after GIMPLE Lowering and Before Register Allocation

At the point RTL is constructed:

- SSA properties have been removed or lowered to assignment form.
- Temporaries and addressing expressions have not yet been lowered into physical registers.
- RTL still contains **pseudo registers**, which will later be mapped to hardware registers or spilled.

Example RTL before register allocation:

```
(set (reg:DI 72) (plus:DI (reg:DI 70) (reg:DI 71)))
```

After register allocation:

```
(set (reg:DI rax) (plus:DI (reg:DI rdx) (reg:DI rcx)))
```

8.1.6 Summary

Component	Role	Impact
RTL Expressions	Abstract machine-level data-flow	Foundation for instruction selection
Operand Constraints	Legality rules for operands	Ensures target-correct encodings
Machine Modes	Encodes width and type	Drives register assignment and instruction variants
Pseudo Registers	Pre-RA value placeholders	Enable register allocation optimization

RTL provides the **bridge between IR-level semantics and target architecture execution requirements**. Operand constraints and mode typing ensure that instruction selection produces **architecturally valid and optimally encodable** machine-level output.

8.2 Constraints (`M`, `r`, `i`, `s`, `g`, `m`, ...): Register vs Memory Operand Legality

Instruction selection in GCC depends on **operand constraints**, which define the set of operand forms that a machine instruction pattern may accept. Constraints appear in **`define_insn`** rules and apply to each operand independently. Their interpretation determines whether the compiler may use a register, immediate constant, or memory reference for a particular operand during instruction matching. Understanding constraint interaction is essential to avoid unintended register pressure, spills, and instruction fallback to slower alternatives.

8.2.1 Constraint Classes and Operand Roles

Constraints control *operand admissibility*, not scheduling or register allocation. Operand matching occurs **before** register allocation; therefore, constraints influence which RTL forms survive into the allocation phase.
Common constraint categories:

Constraint	Operand Class	Meaning
`r`	Register operand	Must be held in a general-purpose register
`m`	Memory operand	Must be a valid memory reference under target addressing rules
`i`	Immediate operand	Compile-time constant encodable in instruction format

Constraint	Operand Class	Meaning
`g`	General operand	Any valid operand: register, memory, or immediate
`s`	Symbolic constant	Link-time-resolvable address or offset
`M`	Memory reference requiring special addressing form	Typically aligned, scalar, or page-based constraints

An instruction pattern may specify multiple constraints to express alternative encodings:

```
"r,m" → operand may be register or memory
```

Constraints govern **legality**; **cost models** influence which alternative is preferred during selection.

8.2.2 Register Operand Constraints (`r` and Register Classes)

The `r` constraint indicates that the operand must reside in a **general-purpose register**. During instruction matching, RTL is rewritten:

```
(set (reg:DI 5) (plus:DI (reg:DI 5) (const_int 4)))
```

If a value is not already in a register, a **reload** is introduced:

```
(reg ← mem)
use reg
```

Constraint refinement using architecture-specific classes enables more granular control:

Constraint	Register Class	Purpose
`"r"`	any GPR	Default integer register operand
`"a"`	accumulator (e.g., x86 `rax`)	Used for multiply/divide forms
`"x"`	SIMD register	Required for vector instruction patterns

Constraint specialization prevents invalid instruction forms during matching.

8.2.3 Memory Operand Constraints (`m` and Sub-Forms)

The `m` constraint indicates that the operand **must** be a loadable memory reference. The target backend determines what constitutes a legal addressing mode:
Example x86-64 addressing RTL:

```
(mem:SI (plus:DI (reg:DI rbx) (const_int 32)))
```

Some architectures support:

- Base + displacement
- Base + index scaling
- PC-relative addressing

Others restrict memory operands to fixed offset or aligned locations.
When constraints disallow memory for an operand, GCC inserts a **temporary register load**:

```
temp = mem(...)
use temp
```

This influences instruction scheduling and register pressure.

8.2.4 Immediate Operand Constraints (`i`, `n`, `I`, `J`, ...)

Immediate constraints enforce **encoding feasibility**, not semantic correctness. For example, on x86-64:

Constraint	Immediate Encoding Rule
`i`	Any integer constant representable in the target mode
`I`	8-bit constant sign-extended into operand width
`J`	`0` only (used for test/compare special forms)

Example:

```
(plus:SI (reg:SI rdi) (const_int 300))
```

If `300` cannot be encoded as an 8-bit signed immediate, a **load-constant+add** sequence is required.

8.2.5 General Operand Constraint (`g`)

The `g` constraint defers operand form selection to the compiler. It indicates that the operand may be a:

- Register
- Memory reference
- Immediate constant

However, it allows **suboptimal instruction sequences** if used indiscriminately. Precision in constraint specification improves register allocation stability and instruction compactness.

8.2.6 Symbol Constraints (`s`)

The `s` constraint permits symbolic constants whose values are resolved by the linker, such as:

```
(symbol_ref:DI "global_array")
```

These often require addressing sequences based on platform relocation models (e.g., ELF GOT and PLT placement on x86-64). Backends use this constraint to ensure correct relocation emission.

8.2.7 Summary

Constraint	Operand Type	Enforcement Effect
`r`	General-purpose register	Triggers reload if value resides in memory
`m`	Memory reference	Disallows register-only instruction forms
`i` / `I` / `J`	Immediate constant classes	Restricts encoding based on architectural limits
`g`	General operand	Least restrictive; can reduce optimization precision
`s`	Link-time symbolic constant	Ensures relocatable addressing correctness
`M`	Special memory class	Matches architecture-specific addressing modes

Operand constraints define the **legal operand shapes** for instruction patterns and thus govern the **space of encodable machine code**. Precise constraint use improves code generation determinism, reduces reload insertion, and enables backend optimizers to preserve intended hardware execution characteristics.

8.3 Machine Pattern Matching and Macro-Op Fusion Candidates

Machine pattern matching is the stage in which RTL expressions are translated to concrete target instructions by matching them against patterns defined in the machine description (MD) files. The **matcher operates before register allocation**, using structural pattern equivalence and operand constraints to select a valid *instruction form* from the available encodings.
For modern superscalar out-of-order architectures, correctness alone is insufficient: the mapping must account for **microarchitectural fusion behavior**, where the CPU may fuse multiple dependent operations into a single internal micro-op. The presence or absence of fusion has measurable effects on pipeline throughput, decode width utilization, and branch prediction cost. Thus, pattern selection incorporates **fusion feasibility** when forming instruction candidates.

8.3.1 MD Pattern Identification

MD patterns describe valid RTL-to-instruction mappings:

```
(define_insn "addsi3"
  [(set (match_operand:SI 0 "register_operand" "=r")
        (plus:SI (match_operand:SI 1 "register_operand" "r")
                 (match_operand:SI 2 "arith_operand"    "rI")))]
  "TARGET_64BIT"
  "add{l}\t%2, %1"
)
```

Pattern matching requires:

1. Structural equality of RTL *operator tree* (`plus`, `set`, etc.)

2. Operand mode consistency (`SI`, `DI`, etc.)

3. Constraint satisfaction for operand forms

4. Optional predicate satisfaction (e.g., target-family check, alignment model)

Patterns are ranked based on constraint specificity, not instruction latency.

8.3.2 Fusion-Friendly Canonical Forms

Modern x86-64 cores support **macro-op fusion**, where certain instruction pairs are merged into a single dispatch/issue micro-op. Fusion reduces front-end pressure and can hide branch latency.
Examples of fusion pairs (Intel and AMD families):

Pair Form	**Fusable Condition**
`cmp reg, reg/imm` + `jcc target`	No intervening instructions; operands in canonical order
`test reg, reg` + `jcc target`	Same
`inc/dec reg` + `jcc target`	Target-dependent; disabled on some microarchitectures
`add/sub reg, imm` + `cmp reg, imm`	Fusion via flag equivalence if CFG allows forward substitution

For the compiler to take advantage of fusion, the pattern matcher must **preserve canonical comparison-and-branch forms**, avoiding transformations that obscure the comparison operation or introduce unnecessary temporaries.
Example canonical RTL suitable for fusion:

```
(set (reg:CC FLAGS)
     (compare:CC (reg:SI rdi) (const_int 0)))
(set (pc)
     (if_then_else (ne (reg:CC FLAGS) (const_int 0)) (label_ref L1) (pc)))
```

If a transform rewrote this into a conditional move or branchless select, fusion becomes unavailable.

8.3.3 Pattern Matching for Fusable RTL Sequences

The matcher does not fuse operations; instead, it selects instruction patterns that **preserve fusion eligibility**. This depends on:

- Avoiding lowering `compare` into a disguised arithmetic instruction unless profitable.
- Ensuring flag-producing instructions are used rather than synthetic boolean temporaries.
- Maintaining direct branch conditions rather than value-based conditional tests.

Example non-fusable form (undesirable):

```
t1 = (x == 0);
if (t1) goto L;
```

RTL lowers to:

```
(set (reg:SI rax) (eq (reg:SI rdi) (const_int 0)))
(set (pc) (if_then_else (ne (reg:SI rax) (const_int 0)) ...))
```

This blocks fusion because the compare is no longer tied to the branch decision.

8.3.4 MD Pattern Encoding for Fusion-Aware Selection

Backends define *fusion-preferred* patterns by ensuring:

- `compare` and conditional branch share the same operand source.
- No spurious moves separate compare and branch.
- Operand constraints prevent unnecessary register reloads that break fusion adjacency.

Example fusion-friendly MD pattern pair:

```
(define_insn "cmp_si"
  [(set (reg:CC FLAGS) (compare:CC (match_operand:SI 0 "register_operand" "r")
                                   (match_operand:SI 1 "nonmemory_operand" "rI")))]
  ""
  "cmp{l}\t%1, %0")

(define_insn "branch_cc"
  [(set (pc)
        (if_then_else (match_operator 0 "comparison_operator"
                                      [(reg:CC FLAGS) (const_int 0)])
                      (label_ref (match_operand 1 "" ""))
                      (pc)))]
  ""
  "j%C0\t%l1")
```

The shared use of `(reg:CC FLAGS)` is critical: it ensures the backend preserves flag dependency, enabling micro-op fusion.

8.3.5 Practical Fusion Limitations

Fusion eligibility may be disabled when:

Limitation Type	Example
Instruction ordering constraints	Scheduling pass separates compare and branch
Register allocation spills	Additional loads inserted between fused pair
Target microarchitecture differences	Some cores disable `INC`/`DEC` + `Jcc` fusion
Mixed-mode arithmetic	Flags not preserved across inserted narrowing/widening ops

Backends therefore place fusion consideration **before** scheduling and after initial RTL formation.

8.3.6 Summary

Stage	Role
Pattern Matching	Maps RTL expressions to legal machine instructions
Constraint Enforcement	Ensures operands are encodable for the chosen pattern
Fusion-Aware Selection	Preserves sequences that allow hardware macro-op fusion

Stage	Role
Backend Machine Model	Determines whether fusion improves decode/dispatch efficiency

Correct pattern selection ensures RTL lowering maintains **microarchitecturally optimal control-flow and flag dependency structure**, enabling the CPU to exploit **macro-op fusion** and reducing execution pipeline pressure.

8.4 RTL Verification Passes and Semantic Equivalence Rules

After RTL generation and machine pattern selection, GCC performs a series of verification and normalization passes to ensure that the RTL representation is well-formed, structurally consistent, and semantically acceptable for register allocation, scheduling, and final assembly emission. These verification passes detect illegal operand modes, invalid addressing forms, incorrect data-dependencies, and transformations that would violate the observable semantics of the original program.
The correctness criteria enforced at this stage are *mechanical*: they do not reconsider language-level meaning but guarantee that the RTL graph preserves program semantics under the machine model.

8.4.1 Structural Well-Formedness Checks

The verifier ensures that each RTL expression satisfies:

1. **Operand Count Validity**:
   The number of operands matches the operation specification (e.g., `set` must have two operands).

2. **Mode Consistency**:
   The mode of the result matches the mode of the operator and its operands unless the operator is explicitly mode-polymorphic.

3. **Legal Operand Class Binding**:
   Operands must satisfy architecture-specific predicate checks (e.g., memory operands must pass `memory_operand_p` under target addressing rules).

4. **No Illegal Nested Sets**:
   RTL prohibits nested assignment expressions inside the left-hand side of a `set` unless handled by multi-output patterns.

Structural correctness must hold *prior* to register allocation or scheduling to avoid invalid machine states.

8.4.2 Data-Dependence and Liveness Preservation

The verifier ensures RTL respects the **Single Assignment + Mutation** model at the register granularity:

- Each pseudo register must be *defined* before first use.
- The live range of registers must not violate block dominance constraints.
- Flag registers and condition codes must not be clobbered without explicit representation.

Incorrect transformations in earlier passes may generate undefined uses:

```
(set (reg:SI 42) (plus:SI (reg:SI 42) (reg:SI 19)))   // illegal: read-before-write
```

This is corrected by insertion of a temporary or by converting to a canonical `set` followed by explicit copy.

8.4.3 Address Legality and Alignment Rules

Memory access expressions are validated against the target's addressing mode semantics:

- Scaled index addressing must match encodable forms.

- Displacements must fit relocatable encoding ranges.
- Misaligned access is permitted only if the architecture supports it or if alignment assumptions have been explicitly relaxed.

Violations trigger the **reload pass**, which rewrites memory expressions into:

```
reg_tmp = address computation
(load/store using reg_tmp)
```

This step is correctness-preserving but may introduce spill pressure.

8.4.4 Semantic Equivalence Constraints

Verification ensures that transformations preserve **observable behavior** under the C++ abstract machine and target ABI:

Category	Constraint	Example Violation
Volatile access	Must not be reordered or eliminated	Removing repeated volatile reads
Strict aliasing	Type-based disambiguation cannot be broken	Combining stores across incompatible pointer types
Floating-point semantics	IEEE exceptions and rounding must be preserved unless explicitly relaxed	Reassociation of FP adds without `-ffast-math`
Atomic operations	Must maintain memory order model	Lowering `seq_cst` fences incorrectly

RTL verification enforces memory ordering through explicit insertion and preservation of memory barriers and mode-specific atomic operations.

8.4.5 RTL Graph Normalization

Before register allocation, RTL is normalized to reduce unnecessary structural variation:

- Convert multi-step address expressions into canonical form.
- Collapse redundant moves (e.g., `(set (reg X) (reg X))`).
- Remove dead assignments detectable through backward liveness analysis.
- Replace target-independent idioms with target-preferred RTL operators (e.g., `neg` vs `sub` from zero).

This improves pattern matching stability and minimizes the probability of generating register pressure spikes.

8.4.6 Summary

Verification Aspect	**Enforcement Target**	**Resulting Guarantee**
Structural validity	RTL node shape and operand form	Ensures expression tree correctness
Data dependency legality	Register liveness and SSA consistency	Prevents undefined or incorrect value flow
Addressing correctness	Target-specific addressing constraints	Ensures encodable load/store operations
Semantic preservation	C++ memory model + FP + aliasing	Guarantees observable correctness

Verification Aspect	**Enforcement Target**	**Resulting Guarantee**
Normalization	Canonical RTL representation	Enables stable instruction selection and scheduling

Verification ensures that RTL remains a **sound and executable low-level IR**, maintaining semantic equivalence to the source program while preserving the necessary structural guarantees for back-end optimization, register allocation, and final assembly emission.

8.5 Examples: Live RTL → Final x86-64 Assembly Correlation

This section illustrates the relationship between **live RTL** after register allocation and the **final machine instructions** emitted for x86-64. The objective is to show how abstract RTL forms are resolved into architecture-specific register selections, addressing modes, and instruction encodings. The examples are representative of GCC 11–14 code generation behavior targeting the System V AMD64 ABI.

8.5.1 Example Source

```
int add_and_scale(int* x, int i) {
    return x[i] * 3 + 5;
}
```

Compile with inspection:

```
g++ -O3 -S -fdump-rtl-expand -fverbose-asm example.cpp
```

8.5.2 Relevant Live RTL (Post-Expand, Pre-RA Simplified)

```
;; load x[i]
(set (reg:SI 66)
     (mem:SI (plus:DI (reg:DI 64) (ashift:DI (reg:DI 65) (const_int 2)))))

;; multiply by 3
(set (reg:SI 67)
     (mult:SI (reg:SI 66) (const_int 3)))

;; add 5
(set (reg:SI 68)
```

```
    (plus:SI (reg:SI 67) (const_int 5)))

;; return value
(set (reg:SI 0) (reg:SI 68))
```

Key observations:

- `reg:DI 64` corresponds to argument `x`
- `reg:DI 65` corresponds to argument `i`
- The addressing `(plus (reg64) (i<<2))` reflects 32-bit element size
- Temporaries 66, 67, 68 are pseudo registers (pre–register allocation)

8.5.3 Register Allocation Assignments (Typical)

```
reg:DI 64 → rdi   ; pointer argument
reg:DI 65 → rsi   ; index argument
reg:SI 66 → eax   ; loaded element
reg:SI 67 → edx   ; multiplied result
reg:SI 68 → eax   ; reused storage (coalescing)
```

Register coalescing reduces live-range overlap, avoiding unnecessary moves.

8.5.4 Final x86-64 Assembly (Representative Output)

```
add_and_scale:
    mov     eax, DWORD PTR [rdi + rsi*4]   # load x[i]
    lea     edx, [rax + rax*2]             # multiply eax by 3
    lea     eax, [rdx + 5]                 # add 5
    ret
```

Instruction rationale:

RTL Operation	Lowered Form	Reason
`mem load`	`mov eax, [rdi + rsi*4]`	Standard scaled-index addressing
`mult by 3`	`lea edx, [rax + rax*2]`	Strength reduction replaces multiply
`add 5`	`lea eax, [rdx + 5]`	LEA used to fold immediate addition
`return`	`ret`	ABI return in `eax`

Notably, GCC emits **`lea`** in place of multiplication where possible, reflecting a backend strength-reduction rule informed by the cost model.

8.5.5 Example With Alias Inhibition vs `restrict`

Without `restrict`:

```
x[i] = x[i] + 1;
```

Result (typical):

```
mov     eax, [rdi + rsi*4]
add     eax, 1
mov     [rdi + rsi*4], eax
```

With `restrict`:

```
add     DWORD PTR [rdi + rsi*4], 1    # no load/store separation required
```

Alias guarantees directly influence RTL → assembly form.

8.5.6 Example With Loop-Carried Induction

Source:

```
int sum(int* x, int n) {
    int s = 0;
    for (int i = 0; i < n; ++i)
        s += x[i];
    return s;
}
```

Vectorization disabled (for clarity):
Final form (representative):

```
sum:
    xor     eax, eax
    xor     ecx, ecx
.L1:
    cmp     ecx, edi
    jge     .L2
    add     eax, DWORD PTR [rsi + rcx*4]
    inc     ecx
    jmp     .L1
.L2:
    ret
```

Key correlations to RTL:

- `ecx` is the **BIV** induction variable from canonicalization.
- `eax` carries **loop-reduced accumulation**.
- Addressing again leverages scaled-index form.

8.5.7 Summary

Stage	Representation	Key Characteristic
Live RTL	Typed operator trees with pseudo registers	Architecture-aware but register-agnostic
Register Allocation	Pseudoreg → physical reg mapping	Live-range coalescing and spill minimization
Final Assembly	Concrete opcodes and addressing forms	Encodes ISA-efficient addressing and ALU operations

The transition from RTL to final machine code is guided by machine modes, operand constraints, alias guarantees, and microarchitectural cost models. The correspondence is **mechanical and traceable**, enabling correctness validation, performance tuning, and architecture-specific optimization reasoning.

Chapter 9

Register Allocation, Spill Minimization, and Scheduling

9.1 Graph Coloring Allocation and Coalescing

Register allocation assigns **pseudo registers** in RTL to a finite set of **physical registers** on the target architecture. GCC applies a variant of **graph coloring register allocation**, combined with **copy coalescing** and **spill minimization**, to map the infinite-register SSA idealization to the fixed register resources of x86-64. The allocator constructs an *interference graph* representing simultaneous live ranges, then attempts to color the graph using the register classes available in the target machine model.

9.1.1 Interference Graph Construction

An interference graph is defined as:

- Each node represents a pseudo register.

- An edge between two nodes indicates both registers are *live at the same time*, thus cannot use the same physical register.

Liveness information is computed using **backward data-flow analysis**:

```
LIVE-IN(block)  = USE(block)   (LIVE-OUT(block) - DEF(block))
LIVE-OUT(block) =  LIVE-IN(successors(block))
```

Two registers `p` and `q` interfere if:

```
 program point k: p  LIVE(k)  q  LIVE(k)
```

Edges are inserted accordingly.
The resulting graph often has higher-degree nodes representing wide live ranges spanning multiple blocks.

9.1.2 Register Classes and Architectural Constraints

x86-64 exposes distinct register classes:

Class	Members	Typical Usage
GPR 64-bit	`rax rbx rcx rdx rsi rdi r8-r15`	Integer arithmetic, addressing
XMM/YMM/ZMM	SIMD registers	Vectorization and FP arithmetic
FLAGS	Implicit condition register	Generated by ALU ops

Graph coloring is performed **per class**, not globally.
A pseudo can only be colored with registers from the class dictated by its use sites and operand constraints.

9.1.3 Graph Coloring Heuristic

GCC applies a **simplify–spill–select** heuristic:

1. **Simplify:**
   Remove nodes with degree $<$ available registers; push to stack.

2. **Spill Candidates:**
   If no such nodes exist, select a spill candidate based on **spill cost**:

   - Estimated dynamic use frequency.
   - Memory access penalties.
   - Loop nesting depth weighting.

3. **Assign Colors (Select):**
   Pop nodes in reverse order, assigning the lowest-cost available register.

The algorithm guarantees coloring when feasible; otherwise, spill insertion rewrites the RTL and the allocator re-runs on the modified graph.

9.1.4 Copy Coalescing

Copy coalescing reduces the number of register move instructions by forcing two pseudo registers to share the same physical register, provided that doing so does not introduce new interference. For:

```
(set (reg p) (reg q))
```

Coalescing attempts to color `p` and `q` identically.
Conditions for coalescing:

- `p` and `q` must not interfere.
- Their live ranges must be merged without exceeding degree thresholds.

When beneficial, coalescing:

- Removes move instructions at instruction selection level.
- Reduces register pressure by tightening live-range boundaries.

9.1.5 Conservative vs Aggressive Coalescing

GCC uses *aggressive coalescing*, relying on later spill decisions to undo pathological merges:

- **Conservative coalescing** performs only safe merges.
- **Aggressive coalescing** merges when beneficial, even if liveness expansion increases temporary node degree.

This strategy leverages the **rematerialization** and **reload** support in later passes to correct adverse cases.

9.1.6 Interaction with SSA Form

While SSA inherently minimizes interference, RTL is *not* SSA.
However, SSA-derived live ranges guide coalescing:

- The allocator attempts to preserve SSA -webs (related SSA names).
- -resolution becomes register coalescing when possible.
- If a -node cannot be coalesced, moves are inserted in predecessor blocks.

Thus, coalescing serves as the **structural analog of SSA -elimination**.

9.1.7 Summary

Component	Purpose	Result
Interference Graph	Encodes simultaneous liveness constraints	Determines register exclusivity
Graph Coloring	Assigns physical registers to pseudos	Produces legal allocation or triggers spill
Spill Heuristic	Minimizes memory overhead	Balances performance vs. register scarcity
Coalescing	Eliminates redundant copies and resolves -nodes	Reduces move instructions and live-range fragmentation

Graph coloring allocation and coalescing ensure that the **abstract SSA machine model** is lowered into an **architecture-valid and performance-efficient** mapping to physical registers, balancing instruction count, memory traffic, and pipeline utilization.

9.2 PBQP Allocation and Hybrid Region Spilling

While graph coloring remains the dominant allocation strategy in GCC's RTL backend, certain register allocation subproblems cannot be solved optimally under classical coloring heuristics without excessive spilling or fragmentation. To address these cases, GCC employs **Partitioned Boolean Quadratic Programming (PBQP)** allocation for constrained regions, particularly those involving complex register classes, vector registers, and instructions with tight operand coupling. PBQP supports approximate, cost-driven register assignment where interactions among live ranges form cost matrices rather than simple interference edges.
Hybrid allocation in GCC combines **graph coloring for general-purpose regions** and **PBQP-based allocation for constrained subregions**, with **region-based spilling** to minimize spill density and memory traffic under register pressure.

9.2.1 PBQP Formulation Overview

In PBQP allocation:

- Each pseudo register corresponds to a **variable**.
- Each allowable register assignment corresponds to an **option** for that variable.
- Each assignment has an associated **base cost**, reflecting spill likelihood, register preference, and operand constraints.
- Interferences and preference interactions are expressed as **pairwise cost matrices**.

Objective:

```
Minimize: Sum(base_cost(p)) + Sum(cost_matrix(p, q))
```

where `p` and `q` are pseudo registers live simultaneously.
Unlike graph coloring, PBQP allows the allocator to choose suboptimal (but globally profitable) assignments to reduce spill propagation and constraint cascades.

9.2.2 Constrained Allocation Scenarios Requiring PBQP

PBQP is invoked when:

1. **Operand classes differ across uses** (e.g., a pseudo register must use a SIMD register for one instruction and a GPR for another).
2. **Instructions enforce register pairing**, such as multiply-high/low or specific operand register combinations.
3. **Vectorized loops** require consistent allocation for lanes to preserve shuffle and reduction patterns.
4. **Coalescing would create dense cliques** that collapse under graph coloring.

PBQP allows the allocator to express these requirements as structured costs rather than binary infeasibility.

9.2.3 Hybrid Region-Based Spilling

Instead of spilling at the whole-function level, GCC performs **region-based spilling**:

- The function is partitioned into *allocation regions* (commonly loops or basic block clusters).
- Each region undergoes allocation independently.
- Spill decisions consider **loop nesting depth**, frequency, and memory bandwidth model.

Spill cost approximation:

```
spill_cost = dynamic_frequency * penalty(memory_latency + pipeline_stall_cost)
```

Spills in hot loops are avoided unless interference pressure exceeds register capacity across all feasible allocations.

9.2.4 Live-Range Splitting under PBQP

PBQP enables precise **live-range splitting**, dividing a pseudo register's lifetime to reduce the interference footprint:

```
p:
split into:
p :
p :
```

This permits:

- Different register assignments for subregions.
- Spill insertion only where pressure peaks.
- Coalescing applied locally rather than globally.

Live-range splitting is essential in vectorized kernels, where accumulator registers conflict with induction variables and address registers.

9.2.5 Interaction with Scheduling and Rematerialization

The allocator communicates spill decisions to the scheduler:

- If an intermediate value is **cheap to recompute**, it is marked **rematerializable** rather than spilled.

- For x86-64, immediates, address constants, and loop-invariant scale factors are prime candidates.

Example canonical rematerialization substitution:

```
spill(reg)
 → (reload cost) > (recompute ALU op)
 → emit recompute instead of load
```

This avoids memory bandwidth penalties in tight loops.

9.2.6 Summary

Component	Role	Benefit
PBQP Allocation	Cost-minimized register assignment under complex constraints	Handles SIMD and operand-class conflicts more effectively than graph coloring
Region-Based Spilling	Localized spill decisions based on hotness and pressure	Minimizes memory traffic in hot loops
Live-Range Splitting	Reduces interference footprint and spill scope	Preserves allocation quality across program regions
Rematerialization	Recomputes cheap expressions instead of loading from memory	Reduces stall and bandwidth pressure

PBQP-based allocation and hybrid region spilling refine register assignment beyond classical graph coloring, enabling GCC to maintain efficient code generation even under

complex operand constraints and high register pressure, especially in **vectorized, loop-intensive, and latency-sensitive kernels**.

9.3 Scheduler: Port Pressure, Latency, Throughput Tables

Instruction scheduling in GCC is performed **after register allocation** to map the final RTL instruction stream onto the target microarchitecture's execution resources. Modern x86-64 processors feature **multiple execution ports**, heterogeneous functional units, pipelined ALUs, SIMD units, and load/store subsystems. The scheduler attempts to minimize **stall cycles** and **execution port contention**, while improving pipeline utilization and reorder buffer stability.
The scheduler uses **machine cost models** derived from the target's instruction tables, including:

- **Latency**: cycles before a result becomes available.
- **Throughput**: cycles per issued instruction under steady state.
- **Port usage mask**: which execution pipelines an instruction may use.
- **Load/store characteristics**: memory latency, forwarding rules, and bandwidth.

These models guide reordering decisions to maintain both **data dependency correctness** and **executability under microarchitectural resource limits**.

9.3.1 Instruction Latency Constraints

Latency defines how many cycles must elapse before an instruction's result can be consumed. The scheduler preserves **true dependencies**:

```
t2 = f(t1)
t3 = g(t2)
```

If `f()` has latency `L_f`, then the scheduler must ensure that `g()` is not issued before `t2` is available. If sufficient **independent instructions** exist, they are scheduled in the gap to avoid pipeline stalls. Failure to find independent work results in a **data hazard stall**.

GCC uses **dependency edge weightings** to prioritize latency-critical paths when selecting scheduling order.

9.3.2 Execution Port Pressure and Resource Contention

Each instruction type maps to one or more execution ports. For example, on a modern Intel core:

Instruction Type	Possible Ports	Notes
Integer ALU `add/sub`	P0, P1	High throughput
Integer multiply	P0	Higher latency
Load	P2/P3	Memory hierarchy dependent
Store	P4 + memory write-back path	Store-buffer capacity limits
SIMD `addps` / `mulps`	P0, P1	Vector throughput dependent on width

The scheduler distributes instructions to **avoid sustained load on a single port**, improving throughput.

This is crucial in **vector-heavy loops**, where unchecked accumulation of loads, stores, or multipliers saturates particular resources.

9.3.3 Throughput-Based Instruction Arrangement

Throughput defines the expected steady-state rate of an instruction sequence. For loop kernels:

```
Throughput = max( port_pressure, dependency_chain_latency, memory_bandwidth_limit )
```

The scheduler works to minimize the dominant term:

- If port pressure dominates → reorder or interleave instruction types.
- If latency dominates → interleave independent operations to hide wait cycles.
- If memory stalls dominate → prefer software prefetching and register-blocking patterns.

Throughput considerations are strongest in loops with **hot execution frequency** determined from profile-guided optimization (PGO) or heuristics.

9.3.4 Scheduling Boundary Constraints

The scheduler must respect:

- **Dependency barriers** (e.g., condition code dependencies, atomic operations)
- **Control-flow boundaries** (cannot reorder across unpredictable branches)
- **Volatile load/store ordering semantics**
- **Memory fence enforcement** under C++ memory model

x86 enforces **strong memory ordering**, but compiler scheduling must maintain ordering semantics when multiple threads observe operations.

9.3.5 Example: Scheduling a Hot Loop Body

Consider:

```
for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + b;
```

Lowered key instructions (vectorized case omitted):

Instruction	Ports	Latency	Throughput Consideration
Load `x[i]`	P2/P3	memory-bound	Interleave loads
Multiply	P0	high latency	Hide with independent loads
Add	P0/P1	cheap	Schedule after multiply to avoid stall
Store `y[i]`	P4	buffer limited	Ensure store buffer not saturated

Optimal scheduling clusters **loads early**, places **multipliers first among ready ops**, and positions **stores last** to minimize pipeline blockage.

9.3.6 Summary

Component	Role	Result
Latency tables	Establish dependency timing	Prevent data hazards and stalls
Port usage maps	Guide instruction interleaving	Reduce execution port contention

Component	Role	Result
Throughput models	Evaluate sustainable pipeline rate	Improve loop performance and pipeline occupancy
Scheduling legality checks	Ensure semantic correctness	Maintain defined C++ memory and ordering behavior

The scheduler transforms a functionally correct sequence of RTL instructions into a sequence that **matches the physical constraints of the execution pipeline**, minimizing cycles lost to stalls and maximizing sustained computational throughput on x86-64 microarchitectures.

9.4 Skylake-Class µArch Execution Ports (0,1,2,3,4,5,6)

Modern x86-64 processors in the **Skylake-class microarchitecture family** (Skylake, Cascade Lake, Coffee Lake, Ice Lake Server with minor variations) use a **distributed execution backend** where instructions are issued to **multiple execution ports**, each associated with one or more functional units. Instruction scheduling in GCC's backend is influenced by these port mappings to avoid **resource saturation**, reduce **critical-path latency**, and optimize **steady-state throughput** in loop kernels.
The port configuration defines **where** each instruction may execute and **how often**, allowing the scheduler to interleave operations to maintain pipeline concurrency.

9.4.1 Execution Port Summary

Port	Primary Functional Units	Operations
Port 0	ALU / Integer Multiply / FP Add	`add`, `sub`, integer multiply, scalar/packed FP add
Port 1	ALU / Integer Multiply / FP Multiply	`add`, integer multiply, scalar/packed FP multiply
Port 2	Load Unit (address generation + load pipeline)	Load from memory
Port 3	Load Unit (dual load pipeline)	Load from memory
Port 4	Store Data + Store Address	Memory stores and store address resolution
Port 5	Branch Unit / Misc	Branches, flag-dependent operations

Port	Primary Functional Units	Operations
Port 6	Address Generation Unit (AGU) support	Complex address formation, interaction with ports 2–4

The execution engine can often **issue multiple µops per cycle**, but port contention occurs if too many operations target the same port class.

9.4.2 ALU and FP Arithmetic Distribution (Ports 0 and 1)

Arithmetic operations—integer or floating-point—are distributed across **Ports 0 and 1**:

- **Integer add/sub** and logic instructions are symmetric across both ports.
- **Integer multiply** may have asymmetric latency and throughput constraints but is still shared across the same two ports.
- Scalar and packed FP addition vs. multiplication are allocated to **separate pipelines**, but still map primarily through Ports 0/1.

Implication for scheduling:

- Interleave add and multiply operations to avoid single-port bottleneck.
- Avoid clustering dependent FP multiplies; latency stacking increases stall probability.

9.4.3 Load and Store Pipelines (Ports 2, 3, 4, 6)

Loads and stores are bandwidth-limited and affect loop performance. The relevant port usage:

Operation	Ports Used	Notes
Load (simple base+index)	Port 2 or Port 3	Two loads may issue per cycle if independent
Store	Port 4	Store data path; constrained by store buffer entries
Complex Addressing (scaled index + displacement)	Port 6 assists address generation	AGU availability influences scheduling order

Scheduling considerations:

- Sustained loops with two loads + one store per iteration saturate **Ports 2/3/4** before arithmetic becomes limiting.
- AGU pressure (Port 6) becomes the bottleneck when using multiple complex addressing forms in vector loops.

9.4.4 Branching and Control Dependencies (Port 5)

Branches and flag-dependent operations execute on **Port 5**, with branch prediction and µop fusion influencing cost:

- `cmp` + `jcc` fusion reduces front-end µop count.
- Divergent branches degrade throughput; masking/vector predication is preferred on AVX-512 systems.

The scheduler attempts to:

- Maintain compare→branch adjacency to enable fusion.
- Avoid unnecessary flag-setting operations that would consume Port 5 bandwidth.

9.4.5 Performance Implications in Loop Kernels

In compute kernels, steady-state throughput is frequently determined by:

```
max(
  ALU/FP port utilization (Ports 0/1),
  Load bandwidth (Ports 2/3),
  Store bandwidth and buffer limits (Port 4),
  Address generation constraints (Port 6)
)
```

Case analysis:

- **Memory-bound loops** saturate Ports 2/3/4 before arithmetic limits are reached.
- **Dot-product or matrix kernels** saturate Ports 0/1 (FP pipelines).
- **Gather/scatter patterns** become Port 6 constrained due to complex addressing.

Thus, GCC's scheduling heuristics bias:

- Out-of-order reordering to hide arithmetic latency.
- Load clustering early in iteration to overlap latency.
- Store deferment to avoid store buffer congestion.

9.4.6 Summary

Port Group	Resource Type	Scheduling Goal
Ports 0 & 1	Integer/FP ALU pipelines	Balance arithmetic workload to prevent single-port overload
Ports 2 & 3	Load pipelines	Spread loads to maintain bandwidth and avoid serialization
Port 4	Store data pipeline	Space stores to respect buffer and retirement bandwidth
Port 6	Address generation	Simplify addressing forms or interleave AGU consumers
Port 5	Branch/control	Preserve fusion patterns and avoid unnecessary flag dependencies

Understanding port usage is essential for correlating **generated assembly** with **actual execution performance**, enabling the compiler backend to emit instruction sequences that align with **Skylake-class superscalar, out-of-order execution characteristics**.

9.5 Examples: Stall Origin Detection via Annotated Disassembly

Performance analysis of generated code requires understanding **where pipeline stalls originate** and whether they are caused by data dependencies, port pressure, memory latency, or speculation failures. Annotated disassembly correlates machine instructions with microarchitectural behavior by overlaying **latency, throughput, and port usage** information on the final binary. This section demonstrates structured stall-source identification for Skylake-class x86-64 hardware.

9.5.1 Example Hot Loop

```
float dot(const float* x, const float* y, int n) {
    float s = 0.f;
    for (int i = 0; i < n; ++i)
        s += x[i] * y[i];
    return s;
}
```

Compiled with:

```
g++ -O3 -march=skylake -fno-tree-vectorize -S dot.cpp
```

Representative assembly (simplified):

```
.L1:
    movss   xmm1, DWORD PTR [rdi + rax*4]      # load x[i]
    movss   xmm2, DWORD PTR [rsi + rax*4]      # load y[i]
    mulss   xmm1, xmm2                         # multiply
    addss   xmm0, xmm1                         # accumulate
```

```
    inc     rax
    cmp     rax, rdx
    jl      .L1
```

9.5.2 Annotated Disassembly with Port Maps and Latency

Applying annotation (via tools such as llvm-mca, IACA-equivalent models, or manual µOp tables):

Instruction	Port(s) Used	Latency	Notes
`movss [rdi+rax*4]`	P2 (Load)	memory-dependent	May stall if L1 miss
`movss [rsi+rax*4]`	P3 (Load)	memory-dependent	Independent load; dual-load bandwidth available
`mulss xmm1, xmm2`	P1 (FP multiply pipe)	~4 cycles	Critical-path contributor
`addss xmm0, xmm1`	P0 (FP add pipe)	~4 cycles	Forced dependency on mulss result
`inc rax`	P0/P1	1 cycle	Not latency-critical
`cmp/jl` (fusable)	P5	1 cycle + branch pred	Macro-op fusion prevents front-end stall

Key performance characteristic:

```
Critical path = mulss → addss dependency chain
```

This chain defines the **minimum achievable throughput** regardless of port availability.

9.5.3 Stall Source Classification

Stalls in this loop may originate from one of four sources:

Stall Type	Detection Pattern	Likely Cause
Memory latency stall	Loads retire slowly; IPC falls toward 1	Input not in L1 cache
Port contention stall	Port 2/3 overcommitted	Excessive load pressure (two loads per iteration)
Dependency stall	FP pipeline throttles	`mulss` latency dominating `addss`
Branch misprediction stall	Loop iteration unpredictable	Data-dependent control flow (not applicable here)

In this kernel, the **dominant limiting factor** depends on data residency:

- If `x` and `y` fit in L1/L2 → dependency-limited.
- If sourced from memory → memory-latency-limited.

9.5.4 Annotated Analysis with Throughput Model

Steady-state throughput estimate (assuming L1 residency):

```
mulss latency  4 cycles
addss latency  4 cycles
Both executed in dependent sequence → throughput limited to ~1 result / 4 cycles
```

Load units (Ports 2 and 3) can sustain **two loads per cycle**, so load bandwidth does not dominate if data is warm.

The backend scheduler **cannot shorten dependency chains**; vectorization is required to reduce dependency depth.

9.5.5 Vectorized Case Contrast (AVX2 / AVX-512)

When vectorized:

```
vmovaps ymm1, YMMWORD PTR [rdi + rax]
vmovaps ymm2, YMMWORD PTR [rsi + rax]
vfmadd231ps ymm0, ymm1, ymm2      # fused multiply-add
```

Characteristics:

- FP multiply and add fused → latency reduced to ~**4 cycles for 8–16 elements** in parallel.
- Critical-path length no longer equals scalar dependency chain.
- If stalls persist, origin shifts to **Port 2/3 memory bandwidth**.

Thus annotated stall diagnosis provides direction:

Dominant Stall	**Optimization Strategy**
Dependency chain	Apply vectorization
Load port pressure	Preload / software prefetch / loop blocking
Store buffer pressure	Delay stores or restructure accumulation
Front-end / decode	Ensure fusion patterns preserved (`cmp+jcc`)

9.5.6 Summary

Analysis Target	What Is Inferred	How Used in Optimization
Port assignment	Detect port saturation	Reorder arithmetic and memory operations
Latency path	Identify critical dependency chains	Encourage vectorization or unrolling
Memory pressure	Detect bandwidth or cache limitations	Apply blocking, prefetching, or layout tuning
Branch fusion state	Determine front-end load	Maintain compare–branch adjacency

Annotated disassembly is a **mechanical diagnostic tool**: it reveals stall sources that are otherwise invisible in pure assembly or high-level IR. This correlation is essential for verifying whether **backend scheduling**, **register allocation**, and **loop structure** align correctly with **Skylake-class pipeline execution constraints**.

Chapter 10

x86-64 SIMD Vectorization and Data Layout

10.1 Vector Instruction Selection (SSE → AVX → AVX2)

SIMD vectorization on x86-64 progresses through several instruction set generations, each expanding register width, datatype support, throughput characteristics, and functional capabilities. GCC's backend selects vector instructions based on:

1. **Available target ISA (`-march=` or `-mavx\*` flags)**
2. **Vectorization profitability heuristics** (iteration count, alignment, memory stride)
3. **Scalar evolution and dependence safety**
4. **Lane width selection for throughput vs. register pressure**

The vectorizer operates on **GIMPLE vector IR** and later lowers to RTL patterns corresponding to **SSE**, **AVX**, or **AVX2** instruction forms. The backend ensures that chosen vector width and instruction variants align with the core's **decode, port, and pipeline constraints**.

10.1.1 SSE (Streaming SIMD Extensions)

- Register width: **bits** (`xmm0-xmm15`)
- Data Types: FP32, FP64, and limited integer operations
- Instruction Semantics: *Register-to-register* operations; loads/stores performed explicitly
- No fused multiply-add support

Example lowering (scalar → SSE):

```
a[i] = b[i] + c[i];
```

May be emitted as:

```
movaps  xmm1, XMMWORD PTR [rdi]    ; load b
movaps  xmm2, XMMWORD PTR [rsi]    ; load c
addps   xmm1, xmm2                 ; vector add
movaps  XMMWORD PTR [rdx], xmm1    ; store result
```

SSE vectorization is latency-limited and does not fully exploit modern port distribution; it is retained mainly for compatibility.

10.1.2 AVX (Advanced Vector Extensions)

- Register width increased to **bits** (`ymm` registers)
- **Three-operand** instruction format (`dest = src0 op src1`)
- Split load/store paths from arithmetic to enhance pipeline utilization
- Still no integer gather/scatter; integer support largely scalarized or emulated

GCC prefers AVX patterns when both input and output vectors are representable as `V8SF / V4DF` etc., and when loop bodies have enough work to amortize transition cost. Example:

```
vmovaps ymm1, YMMWORD PTR [rdi]
vmovaps ymm2, YMMWORD PTR [rsi]
vaddps  ymm1, ymm1, ymm2
vmovaps YMMWORD PTR [rdx], ymm1
```

The **VEX prefix** avoids partial register stalls that SSE inflicted when mixed with AVX state.

10.1.3 AVX2 (Integer Vectorization Extension)

AVX2 adds:

- **Full integer ALU support** in 256-bit vectors
- Vector load/store addressing modes equivalent to scalar counterparts
- **Gather instructions** for irregular indexing patterns (`vgatherdps`, `vgatherdpd`)

This enables efficient vectorization of mixed integer-floating kernels, hash functions, pixel and DSP processing, and polynomial arithmetic.
Example:

```
for (int i = 0; i < n; ++i)
    a[i] = b[i] * c[i];
```

Lowered to AVX2 integer multiply:

```
vmovdqu ymm1, YMMWORD PTR [rdi]
vmovdqu ymm2, YMMWORD PTR [rsi]
vpmulld ymm1, ymm1, ymm2     ; 8×32-bit integer multiply
vmovdqu YMMWORD PTR [rdx], ymm1
```

10.1.4 Vector Width and Microarchitectural Throughput

ISA	Register Width	FP Throughput Characteristics	Integer SIMD Capability
SSE	128-bit	Shared FP port, add/mul pairing limited	Partial
AVX	256-bit	Add/mul split pipelines; better ILP	Limited integer
AVX2	256-bit	Same FP model; adds full integer ALU	Full 32-bit + partial 64-bit

Selecting vector width involves:

- **Port utilization analysis** (Ports 0/1 for FP, 2/3 load, 4 store)
- **Load/store alignment** and AGU pressure

- **Register pressure tradeoff** (wider vectors imply fewer registers available for temporaries)

10.1.5 ISA Transition and Domain Penalties

Mixing SSE and AVX instructions may generate **state transitions** incurring performance penalties:

- Transition from SSE to AVX incurs **upper-state zeroing cost**
- GCC avoids mixing domains unless register pressure or ABI constraints force fallback
- Full AVX register domain is preferred for any loop with vector arithmetic

To enforce AVX-domain consistency:

```
-mprefer-vector-width=256
```

For scalar fallback or power-aware behavior:

```
-mprefer-vector-width=128
```

10.1.6 Summary

ISA	Key Benefit	Backend Selection Driver
SSE	Compatibility and minimal register state	Legacy or narrow vector hot paths
AVX	Higher ILP and three-operand form	FP kernels, vector adds/muls

ISA	Key Benefit	Backend Selection Driver
AVX2	Full integer + FP SIMD support	Mixed arithmetic and throughput-critical loops

Vector instruction selection determines **which hardware pipelines** are engaged, **how data is grouped**, and **what degree of parallelism** the loop can sustain. GCC's backend chooses the SIMD width and instruction form that maximize throughput while preserving correctness and respecting microarchitectural scheduling constraints.

10.2 Load/Store Alignment Constraints and Gather/Scatter Costs

Vector performance on x86-64 is strongly influenced by **memory access alignment** and the **regularity** of data indexing. The backend must select load/store forms and evaluate whether irregular memory access patterns require **gather/scatter** operations, register shuffle sequences, or scalar fallback. These choices directly affect latency, throughput, and effective memory bandwidth.

10.2.1 Alignment Constraints for SIMD Loads and Stores

For **SSE**, **AVX**, and **AVX2**, aligned accesses (aligned to the full vector width) are optimal:

Vector ISA	Register Width	Optimal Alignment Boundary
SSE	128 bits	16 bytes
AVX/AVX2	256 bits	32 bytes

Unaligned loads and stores are permissible under all post-Nehalem microarchitectures; however:

- **Aligned loads** typically have identical latency to unaligned loads *when memory is L1-resident.*
- For **cross-cacheline**, **TLB-miss**, or **page-boundary** cases:
  - Unaligned accesses may require two cache-line requests or replays.

- The penalty amplifies in bandwidth-bound loops.

GCC uses alignment inference from:

- Scalar evolution offsets
- Pointer provenance analysis
- Attribute annotations such as `__builtin_assume_aligned` or `alignas`

If alignment can be proven, the vectorizer emits aligned memory instructions:

```
vmovaps ymm0, YMMWORD PTR [rdi]    ; aligned load
```

Otherwise:

```
vmovups ymm0, YMMWORD PTR [rdi]    ; unaligned load
```

The distinction affects replay rates under heavy load pressure.

10.2.2 Stride and Interleave Effects on Access Form

Unit stride (`a[i]`, `b[i]`) produces efficient wide loads/stores.
If **stride > 1** (e.g., `a[i*2]`), the vectorizer attempts:

1. **Scaled addressing**, if legal:

   ```
   vmovaps ymm0, YMMWORD PTR [rdi + rax*8]
   ```

2. **Lane unpacking + permutation**, if stride is affine:

   - Adds shuffle overhead but still cheaper than gather.

If stride is **non-affine** or depends on elements:

```
a[i] = b[index[i]];
```

access becomes irregular.

10.2.3 Gather and Scatter Instructions (AVX2)

AVX2 introduces **gather** operations:

```
vgatherdps ymm0, [rdi + ymm1*4], ymm2
```

Characteristics:

Property	Behavior
Latency	~10–20 cycles (microarchitecture dependent)
Throughput	Effectively serialized for most index patterns
Port Usage	Uses load ports + AGU + retirement resources
Load Granularity	Loads per element, not per vector

Thus, gather performance more closely resembles **scalar loads in parallel**, not one vector load.
The scheduler treats gather as a **composite load** with combined Port 2/3 + Port 6 demand.
Scatter (store equivalent) relies on store buffer capacity and retires at similar cost.

10.2.4 Vectorizer Decision Rules for Gather/Scatter Emission

The vectorizer will emit gather/scatter only if:

1. **Loop trip count is high enough** to amortize the instruction overhead.
2. **Index array is proven non-aliasing**, to avoid additional speculation stalls.
3. The memory footprint is expected to remain **L1/L2 resident**, or prefetching is applicable.

If these conditions do not hold, GCC falls back to:

- **Scalarized load/store sequences**, or
- **Strip-mine and pack**, converting irregular memory to temporary contiguous buffers.

This trade-off is determined by a profitability model evaluating:

```
gather_cost   unrolled_scalar_cost + (register_shuffle_cost × vector_width)
```

10.2.5 Hybrid Approaches: Load + Shuffle vs. Gather

For moderate irregularity patterns (e.g., permutations from small lookup tables):

- The backend attempts **load + `vpshufd` / `vpermps`** sequences.
- Shuffles are handled on **Port 5 or SIMD pipelines**, avoiding load port saturation.
- Shuffles cost ~**1 cycle throughput**, significantly cheaper than gather.

This is preferred when index patterns are **static or compile-time analyzable**.

10.2.6 Summary

Access Pattern	Backend Strategy	Performance Outcome
Contiguous aligned	Use aligned loads/stores	Highest throughput, lowest replay penalty

Access Pattern	Backend Strategy	Performance Outcome
Contiguous unaligned	Use unaligned loads/stores	Slight penalty; minimal if L1-resident
Affine stride	Use scaled addressing + shuffles	Moderate cost, still vector-efficient
Fully irregular	Emit gather/scatter or scalar fallback	High latency; often memory-bound
Static permutation	Load + shuffle instructions	Avoids gather, preserves bandwidth

Alignment and access regularity determine whether vectorization produces **memory-efficient SIMD kernels** or becomes **latency-bound by gather/scatter operations**. GCC's backend selects the vector load/store strategy that minimizes replay, port saturation, and AGU contention while preserving correctness.

10.3 Data Structure Layout for Cache-Optimized Iteration

The effectiveness of SIMD vectorization depends not only on the arithmetic instruction sequence but also on the **spatial and temporal locality** of data. The backend ultimately consumes data in a hardware-defined streaming model, and the layout of arrays, structs, and object aggregates directly influences cache footprint, memory bandwidth, TLB behavior, and alignment guarantees. GCC's vectorization heuristics assume that memory accesses follow **unit-stride, contiguous iteration** where elements required for computation are laid out sequentially in memory.
When data layout does not satisfy these properties, vectorization either fails, requires gather/scatter, or is forced to insert additional shuffle stages, increasing port pressure and cycle cost. Therefore, designing data structures for **predictable linear access** is a prerequisite for generating high-throughput vector code.

10.3.1 AoS vs. SoA Transformations

The difference between **Array of Structures (AoS)** and **Structure of Arrays (SoA)** is fundamental for SIMD.
Example AoS:

```
struct Point { float x, y, z; };
Point a[N];
```

Accessing `a[i].x` yields stride = 3 * sizeof(float) = non-unit stride → requires shuffle or gather.
Equivalent SoA:

```
struct Points { float x[N], y[N], z[N]; };
```

Now `x[i]`, `y[i]`, `z[i]` are all **unit-stride contiguous** → directly vector-loadable.

Layout	Vectorization Suitability	Reason
AoS	Poor	Fields interleaved → non-contiguous per component
SoA	Excellent	Each field forms a contiguous vector array

The compiler cannot always legally convert AoS to SoA; therefore, layout should be determined at design time for bandwidth-critical loops.

10.3.2 Padding, Alignment, and Page-Locality Considerations

Data structures should be aligned to the **vector register width** of the target architecture:

Target ISA	Preferred Alignment
SSE	16 bytes
AVX/AVX2	32 bytes
AVX-512	64 bytes

Misalignment does not cause correctness issues in modern cores, but it increases:

- Load pipeline replay rates
- Data TLB lookup frequency
- Cache-line doubling on boundary crossings

Explicit alignment:

```
alignas(32) float x[N];
```

or dynamic alignment assertions:

```
float* x = static_cast<float*>(aligned_alloc(32, N * sizeof(float)));
```

improve streaming efficiency.

10.3.3 Loop Nest and Tile Layout for Cache Blocking

For multidimensional data (matrices, tensor blocks), iteration should follow **row-major or column-major order** depending on memory layout. Consider row-major storage:

```
for (i)
  for (j)
    A[i][j] = ...
```

This order yields unit-stride iteration in `j`. Reversing the loops introduces **cache line thrashing** and inhibits vectorization because each iteration jumps by one full row.
For larger workloads, **blocking** improves locality:

```
for (ii = 0; ii < N; ii += B)
  for (jj = 0; jj < M; jj += B)
    for (i = ii; i < ii+B; ++i)
      for (j = jj; j < jj+B; ++j)
        A[i][j] = ...
```

Block size `B` is chosen based on L1/L2 capacity and vector width constraints.

10.3.4 Struct Reordering and False-Sharing Avoidance

When structures contain fields with differing update frequencies, **group high-access fields together** to minimize unnecessary cache line loads.
Example:

```
struct S {
    float temperature;
    float pressure;
    int   id;          // rarely used
    float density;
};
```

Reorder to:

```
struct S {
    float temperature;
    float pressure;
    float density;
    int   id;
};
```

This reduces wasted cache bandwidth when streaming the float fields across SIMD operations.
Similarly, in multi-threaded contexts, avoid placing mutable fields accessed by multiple cores on the same cache line to prevent **false sharing**. Use padding:

```
struct alignas(64) Shared {
    float value;
    char pad[60];
};
```

10.3.5 Alignment Propagation Through the Compiler

For the compiler to leverage alignment in vector loads:

- Alignment must be **provable** at compile time (static alignment), or
- Guaranteed via explicit assertion (`__builtin_assume_aligned`).

Example:

```
void f(float* __restrict x) {
    x = (float*)__builtin_assume_aligned(x, 32);
    // vectorizable loop follows
}
```

This allows the backend to emit `vmovaps` instead of `vmovups`, reducing memory replay penalties.

10.3.6 Summary

Data Layout Property	Impact on Vectorization	Resulting Performance Behavior
Contiguous per-field arrays (SoA)	Enables direct vector loads	High throughput, minimal shuffles
Interleaved fields (AoS)	Requires gather/shuffle	Increased latency and port pressure
Alignment to vector width	Reduces load/store replay	Stable streaming bandwidth

Data Layout Property	**Impact on Vectorization**	**Resulting Performance Behavior**
Correct loop iteration order	Preserves spatial locality	Avoids cache thrashing and pipeline stalls
Blocking/tiling	Improves L1/L2 reuse	Helps both scalar and vectorized kernels

Effective SIMD performance depends on designing data layouts that match **hardware memory semantics**, ensuring that **iteration patterns are predictable, contiguous, and aligned**, and avoiding unnecessary addressing complexity.

10.4 ABI Implications of Vector Calling Conventions

Vectorization affects not only computation inside a function but also how data is passed **across function boundaries**. The x86-64 System V ABI (used by Linux) defines how **vector registers**, **aggregate types**, and **SIMD values** are passed between caller and callee. GCC must honor these rules during IR lowering and register allocation to ensure **binary compatibility**, especially across:

- Shared libraries
- Dynamically loaded modules
- Handwritten assembly routines
- Cross-compiler builds (e.g., GCC Clang interoperability)

Thus, SIMD-aware code must consider both the **performance model** and the **ABI stability requirements**.

10.4.1 Vector Registers in the x86-64 System V ABI

Under System V AMD64 ABI:

- **XMM0–XMM7** are used for passing floating-point arguments.
- **XMM0** is used for returning floating-point and vector values up to 128 bits.
- `YMM` registers **do not expand the ABI**; AVX and AVX2 are treated as *extensions* over XMM state.

This means:

- A 256-bit `__m256` function argument is passed **in XMM registers**, split across pairs of registers in the calling convention.
- The caller and callee must preserve **legacy XMM register save rules** even when operating in AVX2 or AVX-512 mode.

Example function:

```
__m256 add_avx(__m256 a, __m256 b) {
    return _mm256_add_ps(a, b);
}
```

The ABI passes `a` and `b` via *expanded XMM register tuples*, not single 256-bit registers at the ABI boundary.

10.4.2 Register Save / Restore Semantics

Registers are classified into:

Register Class	Caller-Saved	Callee-Saved
XMM0–XMM15	Yes	No
YMM upper halves	Yes	No

Thus:

- Functions using vector registers **must not assume registers are preserved across calls**.
- The compiler inserts spills/restores when calling other functions mid-vector loop.
- Deeply vectorized kernels avoid interprocedural calls to prevent **register pressure expansion**.

This influences the **inlining heuristics**: the compiler aggressively inlines vector loops to avoid ABI register crossing overhead.

10.4.3 ABI and State Transition Costs (SSE AVX)

Mixing SSE and AVX instructions causes **state transitions** between legacy XMM state and AVX YMM state. These transitions incur a **performance penalty** on most Intel architectures before Ice Lake.
The ABI enforces:

- Function entry into AVX state using `vzeroupper` to avoid false dependencies in XMM register rename tracking.
- GCC automatically emits `vzeroupper` when a function emits AVX instructions but may be called from a non-AVX caller.

This is required to avoid **register bank stalls**, not for semantic correctness.

10.4.4 Struct and Aggregate Passing Rules

The ABI breaks large composite parameter types into:

- **Base integer arguments** in GPRs
- **Base floating arguments** in XMM registers
- Remaining components **by memory reference**

Example:

```
struct Vec {
    float a[8];    // 8 floats = 32 bytes
};
```

Passing `Vec` by value:

- Does **not** allow direct use of YMM registers at ABI boundary.
- Compiler spills struct to stack, then loads into YMM inside callee.

This means performance-sensitive vector data should be:

1. Passed as **pointers**, not by value.
2. Stored in **contiguous aligned arrays**, not embedded arrays inside aggregates.

10.4.5 Cross-Module Optimization Boundary

When functions are compiled separately:

- ABI boundaries prevent vector-width assumptions.
- Inter-procedural vectorization (IFV) is disabled unless Link-Time Optimization (LTO) is used.

With LTO:

- GCC may perform **cross-module loop fusion and SIMD promotion**.
- Without LTO, vector width decisions are **locally constrained**.

Thus, for high-performance code:

```
g++ -O3 -march=skylake -flto
```

ensures global vector width consistency.

10.4.6 Summary

ABI Rule	Effect on SIMD Code	Practical Guideline
Passing SIMD values follows XMM-based conventions	YMM data must be reconstructed in callee	Prefer passing pointers instead of value aggregates
XMM/YMM registers are caller-saved	Function calls inside vector loops cause register spills	Inline hot loops to avoid ABI crossings
`vzeroupper` required for transition avoidance	Avoid mixing AVX and SSE domains	Compile translation units uniformly with AVX enabled
Struct-by-value blocks vectorization at boundary	Memory copies inserted by ABI rules	Use SoA layout and pointer/passed-by-reference parameters

Correct SIMD usage requires respecting ABI-defined **calling and state-transition rules**, otherwise the backend is forced to insert spills, alignment fixups, or state synchronization operations that can dominate execution cost. Effective vectorized system design therefore couples **data layout decisions** with **function boundary management** and **linkage-time optimizations**.

10.5 Examples: Loop Rewritten into Full AVX2 Pipeline

This section presents a complete example demonstrating how GCC transforms a scalar loop into a fully pipelined **AVX2** vector loop. The objective is to illustrate **IR lowering**, **vector width selection**, **load/store emission**, and **fused arithmetic sequencing** with alignment and scheduling considerations applied.
The example is representative of typical numeric kernels found in linear algebra, DSP, and machine learning preprocessing pipelines.

10.5.1 Original Scalar Code

```
float saxpy(float* __restrict y, const float* __restrict x, float a, int n) {
    for (int i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
    return y[0];
}
```

Memory access:

- `x` and `y` accessed linearly (unit stride).
- No aliasing constraints due to `__restrict`.
- Arithmetic pattern matches **FMA** opportunities when enabled.

10.5.2 GCC Vectorization Conditions

GCC vectorizes the loop if:

1. The loop trip count is unknown but assumed sufficiently large.

2. Alignment constraints allow either `vmovaps` or `vmovups`.

3. The scalar expression can be lowered to **vector multiply-add**.

With:

```
g++ -O3 -march=skylake -ffast-math
```

GCC generates AVX2 vector code due to:

- Contiguous float arrays → eligible for 256-bit vectorization
- Multiply-add pattern → convertible to FMA (`vfmadd231ps`)
- Restrict-qualified pointers → no alias-based dependency hazard

10.5.3 Representative Vectorized Assembly (Simplified)

```
.Lloop:
    vmovups ymm0, YMMWORD PTR [rsi + rax]      # load x[i..i+7]
    vmovups ymm1, YMMWORD PTR [rdi + rax]      # load y[i..i+7]
    vfmadd231ps ymm1, ymm0, ymm2               # ymm1 = ymm1 + ymm0 * ymm2
    vmovups YMMWORD PTR [rdi + rax], ymm1      # store result
    add       rax, 32                           # advance by 8 floats
    cmp       rax, rdx
    jb        .Lloop
```

Where:

- `ymm2` holds broadcasted scalar `a`.
- `rax` increments by **vector width** $\times$ **sizeof(float)** $= 8 \times 4 = 32$ bytes.
- Operations map primarily to:

- Ports 0/1 for FP arithmetic
- Ports 2/3 for loads
- Port 4 for stores

This mapping allows **simultaneous arithmetic and memory movement**, enabling high throughput.

10.5.4 Pipeline Characteristics on Skylake-Class Cores

Component	Behavior
`vmovups` loads	Issue per-cycle on Ports 2 and 3
`vfmadd231ps`	Uses FP add and multiply units; 1 fused op per cycle throughput
`vmovups` store	Uses Port 4 + store buffer, max sustained ~1 store/cycle
Loop update and compare	Branch unit (Port 5), predictable loop — no misprediction penalty

Critical throughput factor:

```
Max throughput  min(
    FP pipeline capacity,
    load bandwidth,
    store bandwidth,
    AGU capacity
)
```

Under L1 residency and steady state, typical throughput approaches:

```
~1 vector FMA per cycle → 8 scalar FMA equivalents per cycle
```

10.5.5 Comparison to Scalar Performance

Implementation	FLOP/iteration	Cycles/iteration (approx.)	Result
Scalar (`mulss` + `addss`)	2	~4–6 cycles	Low throughput, dependency-bound
SSE (128-bit)	8	~2–3 cycles	Improved ILP, limited width
AVX2 (256-bit FMA)	**16**	**~1 cycle steady state**	**~8–10× speedup vs scalar**

Note: Actual speedup depends on memory bandwidth and dataset size.

10.5.6 Observations from Annotated Disassembly

- The compiler **hoists scalar `a`** into a broadcast register once → avoids per-iteration load.
- No gather/scatter required due to **linear access**.
- No register spills occur → register allocation was sufficient.
- No alignment directives emitted → backend assumed unaligned safe (`vmovups`) because input alignment could not be proven.

Performance improves further if alignment is asserted:

```
x = (float*)__builtin_assume_aligned(x, 32);
y = (float*)__builtin_assume_aligned(y, 32);
```

Which enables:

```
vmovaps / vmovaps
```

reducing replay stalls.

10.5.7 Summary

Property	Impact on Vectorization
Unit-stride contiguous arrays	Allows direct vector loads/stores
Restrict-qualified pointers	Enables safe FMA fusion and scheduling
FMA instruction availability	Reduces dependency chain depth
No aliasing / no gather	Avoids pipeline serialization
Alignment proven (optional)	Reduces load/store replay penalty

The rewritten AVX2 loop demonstrates how GCC emits **fully pipelined SIMD instructions** with minimal register pressure and optimal arithmetic/memory pairing. The transformation converts **scalar per-element computation** into **wide parallel execution** that saturates the floating-point and memory execution pipelines efficiently.

Part V

C++ OBJECT MODEL AND RUNTIME ABI

Chapter 11

Itanium ABI Deep Structure for C++

11.1 Symbol Mangling Encoding Structures

C++ symbol mangling under the **Itanium C++ ABI** provides a deterministic encoding for names, types, scopes, templates, and calling conventions, allowing **linkers**, **shared object loaders**, and **debuggers** to operate across languages, compilers, and binary formats. GCC adheres to this ABI for ELF-based systems on x86-64 Linux, ensuring **binary compatibility** across translation units and across compilers conforming to the same ABI contract.
Unlike C, where function names map directly to linker symbols, C++ requires encoding of:

- Namespace and class scopes
- Overloaded function signatures

- Template parameters and instantiations
- Type qualifiers (`const`, `volatile`, reference types)
- Calling conventions and linkage attributes
- Operator names and internal compiler-generated entities

The mangling scheme is lexical and hierarchical, reflecting the **fully qualified signature** of an entity. This creates a **canonical identifier** for each object or function, serving as the program's external identity in the symbol table.

11.1.1 Top-Level Mangling Prefix

All Itanium ABI-mangled C++ symbols begin with:

```
_Z
```

This distinguishes C++ ABI-mangled symbols from C identifiers and other foreign-format symbols in ELF object files.
Example:

```
int add(int, int);
```

Mangled:

```
_Z3addii
```

Here:

```
_Z      # Itanium ABI prefix
3add    # identifier "add" with length 3
ii      # argument types: int, int
```

11.1.2 Name Scoping Encoding

Nested scopes are encoded through **length-prefixed segments**:

```
namespace N { struct S { void f(); }; }
```

Mangled:

```
_ZN1N1S1fEv
```

Breakdown:

Fragment	Meaning
`_Z`	Itanium ABI prefix
`N … E`	Nested name sequence
`1N`	Namespace `N`
`1S`	Class `S`
`1f`	Method `f`
`Ev`	`void` parameter list

This structure allows arbitrary nesting depth without ambiguity.

11.1.3 Type Encoding and Qualifiers

Types are encoded using single-letter codes with modifier prefixes:

Type	Code
`int`	`i`

Type	Code
`float`	`f`
`double`	`d`
`void`	`v`
pointer to `T`	`P <T>`
reference to `T`	`R <T>`
const-qualified `T`	`K <T>`

Example:

```
double* const& g(float);
```

Mangled:

```
_Z1gRfKPd
```

Which expands to:

```
g( float ) returning reference to const pointer to double
```

11.1.4 Template Argument Encoding

Templates introduce parameterized symbols. Each parameter is encoded recursively. Example:

```
std::vector<int>
```

Contains:

```
St6vectorIiSaIiEE
```

Where:

Component	Meaning
`St`	Standard library vendor prefix
`6vector`	Identifier “vector” (length 6)
`I … E`	Template argument list
`i`	`int`
`SaIiE`	`std::allocator<int>`

For nested templates, the encoding compresses repeated sequences using **substitution tables** to avoid duplication and reduce symbol length.

11.1.5 Operator and Special Function Mangling

Operators and special member functions have reserved encodings:

C++ Feature	Mangled Form
Constructor	`C1`, `C2` (in-place and complete object forms)
Destructor	`D1`, `D2`
`operator+`	`pl`
`operator<<`	`ls`
Conversion operator (`operator T`)	`cv <T>`

Example:

```
S::operator int() const;
```

Mangled:

```
_ZNK1ScviiEv
```

11.1.6 Summary

Structural Element	Encoding Mechanism	Purpose
Prefix `_Z`	Identifies Itanium C++ ABI symbol	Distinguishes from unmangled C
Nested names	Length-prefixed scope encoding	Resolves overload and namespace identity
Types and qualifiers	Single-letter encodings with modifier prefixes	Achieves compact and deterministic substitution
Templates	Bracketed argument lists with substitution	Supports parameterized and recursive type systems
Operators and special functions	Reserved encodings	Preserves semantic mapping in binary interface

Symbol mangling is therefore not an implementation detail but a **core ABI mechanism** that ensures **link-time correctness**, **cross-module interoperability**, and **runtime symbol resolution stability** across compiler versions and standard library implementations adhering to the Itanium ABI.

11.2 VTable Encoding, Virtual Base Pointer Offsets, and Thunks

In the Itanium C++ ABI, the representation of polymorphic classes is explicitly defined to guarantee **binary compatibility across compiler versions and compilation units**. The ABI specifies the **vtable layout**, the handling of **virtual base offsets**, and the insertion of **thunks** when dynamic dispatch requires adjustment to either the `this` pointer or the call target. GCC conforms strictly to these rules to ensure that dynamic binding, RTTI lookups, and cross-module polymorphic behavior are stable and deterministic.

11.2.1 VTable Structural Layout

A vtable is an array of function pointers and metadata. For each polymorphic class type, the primary vtable associated with its most-derived object layout contains:

Slot Offset	Entry	Meaning
-2	Address of RTTI object	Used for `dynamic_cast` and type identification
-1	Offset-to-top (for subobject adjustment)	Adjustment needed to recover base from subobject
0..n	Virtual function pointers	Target addresses for dynamic dispatch

The **offset-to-top** field enables recovery of the **complete object pointer** when dispatching a call on a subobject for cases involving multiple inheritance.
Example conceptual structure:

```
vtable:
```

```
[ -2 ] -> RTTI descriptor
[ -1 ] -> offset to most-derived type base
[  0 ] -> &Derived::f
[  1 ] -> &Derived::g
...
```

This structure allows the runtime to restore the correct `this` pointer before invoking a method.

11.2.2 Virtual Base Pointer Offsets (vbpointers)

For classes using **virtual inheritance**, the object representation contains a **virtual base table pointer (vbptr)** referencing a **virtual base offset table**. This table allows determining the physical address of shared virtual base subobjects.
Example:

```
struct A { int x; };
struct B : virtual A {};
struct C : virtual A {};
struct D : B, C {};
```

Class `D` contains only one `A`, shared through virtual inheritance. The vbpointer logic ensures that both `B` and `C` subobjects compute the same physical location for `A`.
The vtable stores:

- The **offset from the current subobject to the virtual base**
- The **offset from the most-derived object to the base**

GCC emits these offsets in the vtable's auxiliary tables, allowing runtime pointer adjustments without performing type-level computation.

11.2.3 Thunks and `this` Pointer Adjustment

A **thunk** is a compiler-generated function whose role is to adjust the `this` pointer and then transfer control to the actual method body. Thunks are required when:

1. A virtual function is inherited through multiple paths and the object layout differs across base subobjects.
2. The function is overridden in a derived class, but dispatch occurs through a base pointer requiring adjustment.
3. Calling conventions or return address conventions require consistency across a hierarchy.

Example pseudo-assembly for a thunk adjusting `this` by a known offset:

```
adjustor_thunk:
    add     rdi, offset_adjustment     ; adjust this pointer
    jmp     Derived::f                 ; tail jump
```

Thunks **do not introduce an additional function frame**. They perform pointer adjustment and branch to the final implementation, ensuring no additional call overhead beyond the required address fixup.

11.2.4 VTable Reuse and Subobject-Specific VTables

A single class may have **multiple vtables** if it appears as multiple subobjects within a hierarchy, especially under repeated and virtual inheritance. The ABI ensures that:

- Each subobject variant has a distinct vtable view.
- The correct vtable is selected at construction time.

- Adjusted function pointers (possibly via thunks) encode the appropriate offset semantics.

The compiler assigns each subobject a unique **vtable address**, enabling RTTI and `dynamic_cast` to determine which subobject layout is active.

11.2.5 Summary of Runtime Dispatch Flow

At the call site for a virtual function:

1. The caller loads the vtable pointer from the object.
2. The appropriate vtable slot entry is selected.
3. If required, a **thunk adjusts** the `this` pointer.
4. Control is transferred to the function implementation.

This process is **O(1)** in runtime complexity and does not require branching over hierarchy depth.

11.2.6 Summary

Feature	Mechanism	Purpose
VTable layout	Offset-to-top + RTTI + dispatch slots	Unified, deterministic polymorphic dispatch
Virtual inheritance offsets	vbptr + offset tables	Shared base resolution without ambiguity

Feature	Mechanism	Purpose
Thunks	`this` pointer adjustment + jump forwarding	Correct dynamic dispatch under non-uniform layouts
Multiple vtable instances	Subobject-specific tables	Represents distinct layout contexts in hierarchies

The Itanium ABI makes object layout and dynamic dispatch **structural**, not heuristic. GCC adheres to these specifications to ensure that polymorphism, downcasting, and cross-module dynamic linkage remain consistent, predictable, and compatible across compilers and shared libraries.

11.3 Exception Table Encoding, DWARF CFI, and LSDA

C++ exceptions on Linux x86-64 under GCC are implemented according to the **Itanium C++ ABI exception propagation model**, which relies on **zero-cost exception frames**, **DWARF Call Frame Information (CFI)**, and the **Language-Specific Data Area (LSDA)** embedded into the binary. The model avoids runtime overhead in non-throwing paths and shifts the cost into the **unwind phase** during exceptional control flow.

This section describes the structure and role of the exception tables and how GCC emits and queries LSDA metadata to perform stack unwinding, destructor invocation, and landing pad selection.

11.3.1 Zero-Cost Exception Handling Model

Unlike setjmp/longjmp-based models, Itanium-style exception handling does not modify the function's normal execution path. Most runtime tables are **read-only data structures** in `.eh_frame` and `.gcc_except_table` sections.

Execution divides into two phases:

1. **Search Phase:**
   The unwinder walks stack frames to identify the handler for the thrown exception. No stack modifications occur yet.

2. **Cleanup Phase:**
   Stack frames between the throw point and handler are unwound; destructors and cleanup blocks are invoked.

Both phases require structured frame metadata to locate saved registers and landing pads.

11.3.2 DWARF CFI and `.eh_frame`

Each function capable of participating in unwinding contains a **Call Frame Information (CFI) record** encoding:

- Canonical frame address (CFA) computation rules
- Saved register locations
- Stack pointer deltas and base pointer restoration instructions

CFI directives are emitted during compilation, often visible in assembly as:

```
.cfi_startproc
.cfi_def_cfa_offset ...
.cfi_offset ...
.cfi_endproc
```

During unwinding, libgcc's **`_Unwind_\*`** routines interpret this metadata to reconstruct register state and return addresses, permitting control flow to move up stack frames safely.

11.3.3 LSDA: Language-Specific Data Area

The **LSDA** augments `.eh_frame` data with **C++-specific exception handling rules**. It resides in the `.gcc_except_table` section and describes:

- The mapping from instruction ranges to **landing pads** (exception handlers or cleanup blocks)

- The types of exceptions matched at each landing pad (decoded via typeinfo pointers)
- Filters and catch-all regions
- Cleanup-only (destructor) regions without catch semantics

The LSDA structure enables the unwinder to determine:

- Whether a frame must participate in unwinding
- Which landing pad to transfer control to when an exception matches

LSDA data is compactly encoded using **DWARF pointer encodings**, relocation types, and PC-range tables.

11.3.4 Action and Call-Site Tables

Within the LSDA:

- A **call-site table** maps instruction ranges to landing pad entry points.
- Each landing pad references an **action table**, describing the exception type sequence to match.

Conceptual form:

```
call-site entry:
    start address
    length
    landing pad offset
    action index
```

Action entries are chained; a negative next-link index terminates the chain.
This allows matching logic:

```
for each type in action chain:
    if thrown_type is derived or equal → handler selected
```

11.3.5 Interaction with `typeinfo` and RTTI Objects

Exception matching relies on comparing the thrown exception's **typeinfo object** pointer with the types listed in the LSDA action table.
This ensures correct selection of:

- Exact match
- Base class match
- `catch(...)` universal handler
- Cleanup-only blocks (for RAII destruction)

RTTI stability across translation units and dynamically loaded libraries is guaranteed due to the ABI's canonical typeinfo uniqueness rules.

11.3.6 Landing Pads and Control Transfer

Landing pads are compiler-generated blocks, not callable functions. A landing pad receives:

- The exception object pointer (`_Unwind_Exception*`)
- The selector value indicating matched handler type index

A typical landing pad structure:

```
.Llanding:
    mov %rdx, %rdi        # move exception pointer
    call _ZdlPv           # invoke destructor if cleanup-only
    jmp  .Lresume         # pass control back to unwinder
```

The unwinder controls PC adjustment and resume execution, preserving full frame semantics.

11.3.7 Summary

Component	Role	Location
DWARF CFI	Defines how to restore frame/register state during unwinding	`.eh_frame`
LSDA	Defines catch/cleanup mapping to code ranges	`.gcc_except_table`
Call-Site Table	Correlates instruction regions to landing pads	Part of LSDA
Action Table	Encodes exception type matching sequence	Part of LSDA
Typeinfo Objects	Identify runtime class relationships	`.rodata` in vtable segments

The Itanium ABI exception model is **structurally deterministic**, separating **mechanical stack recovery** (CFI) from **C++-specific handler semantics** (LSDA), enabling GCC to implement **zero-cost exception dispatch** consistent across dynamic linking boundaries and heterogeneous compilation environments.

11.4 RTTI and Dynamic Type Resolution Through Typeinfo Graph

Runtime Type Information (RTTI) under the Itanium C++ ABI is implemented through a **canonical typeinfo object graph** that describes the dynamic type relationships of polymorphic classes. This enables safe `dynamic_cast`, runtime type comparisons, and exception type matching across **shared libraries**, **translation units**, and **compilers** adhering to the ABI. GCC emits and queries these typeinfo structures at runtime to resolve type identity and inheritance paths without relying on language-level metadata lookup or compiler-generated RTTI registries.

11.4.1 Typeinfo Object Structure

Each polymorphic class has a unique **`typeinfo`** object, generated in the `.rodata` segment. The base representation is defined by:

```
struct __cxxabiv1::__class_type_info {
    const typeinfo* typeinfo;  // pointer to vtable for typeinfo
    const char* name;          // mangled class name
};
```

Derived typeinfo classes extend this to encode **inheritance semantics**. For class types, the ABI defines:

```
struct __cxxabiv1::__si_class_type_info
    : __class_type_info {
    const __class_type_info* base;    // single inheritance base
};
```

For multiple or virtual inheritance:

```
struct __cxxabiv1::__vmi_class_type_info
    : __class_type_info {
    unsigned int flags;               // inheritance characteristics
    unsigned int base_count;
    struct {
        const __class_type_info* base;
        long offset_flags;            // offset + virtual inheritance indicators
    } base_info[];
};
```

Thus, RTTI encodes both **type identity** and **topological inheritance structure**, forming a **type graph**, not a flat hierarchy.

11.4.2 Canonical Uniqueness and Linkage Consistency

To avoid ambiguities during dynamic linking:

- Each class produces exactly **one canonical typeinfo object**.
- Shared libraries export typeinfo symbols with **weak linkage**, allowing the dynamic loader to perform **pointer identity coalescing**.
- Two types are considered the same at runtime **if and only if their typeinfo pointers compare equal**.

This pointer identity rule ensures that `dynamic_cast` works even when classes are defined across multiple shared objects compiled separately.

11.4.3 Dynamic Type Resolution Algorithm (`dynamic_cast`)

To perform a downcast or cross-cast, GCC utilizes:

1. The **vtable pointer** of the dynamic object to obtain the RTTI pointer of the most-derived type.
2. The **typeinfo graph** to search for the requested target type.
3. The **offset_flags** values to compute the correct `this` pointer adjustment if necessary.

The resolution follows these rules:

Case	Outcome
Target type is a public unique base	Return adjusted pointer
Target type appears multiple times via virtual inheritance	Resolve to shared virtual base
Target type is inaccessible or ambiguous	Result is null pointer for pointer cast; throws `bad_cast` for reference cast

Resolution is structural and derivation-based, not textual or linkage-name based.

11.4.4 Using RTTI in Exception Matching

During exception propagation, the LSDA lists **typeinfo pointers** describing catch patterns. Exception matching operates as:

```
if (thrown->typeinfo == catch_typeinfo)
    match;
else if (thrown->typeinfo is derived from catch_typeinfo)
    match;
else
    continue search;
```

This is identical to `dynamic_cast` relationship matching and relies on the same typeinfo graph traversal.

11.4.5 Example: Multiple and Virtual Inheritance Type Resolution

```
struct A { virtual ~A() {} };
struct B : virtual A {};
struct C : virtual A {};
struct D : B, C {};
```

RTTI graph relationships:

```
D → (B, C) → A
```

Since `A` is a **virtual base**, all paths must converge to a **single shared subobject**.
The typeinfo for `D` contains:

- An entry marking `A` as virtual
- Offset metadata instructing the runtime how to locate the **unique `A` subobject**

During:

```
A* pa = dynamic_cast<A*>(static_cast<D*>(p));
```

The runtime:

1. Identifies that `A` is a virtual base of `D`.
2. Reads `offset_flags` from typeinfo.
3. Adjusts the pointer to yield the correct shared base address.

No runtime scanning of object memory occurs; adjustment uses static layout metadata.

11.4.6 Summary

Component	Role	Stability and Guarantee
`typeinfo` objects	Identify class types	Canonical and unique across shared objects
Inheritance descriptors	Encode class hierarchy structure	Fully structural, no reliance on reflection systems
`dynamic_cast` resolution	Computes correct adjusted pointer	Deterministic graph walk using offset encodings
Exception matching	Uses same typeinfo graph for handler selection	Unifies RTTI and exception semantics

RTTI in the Itanium ABI is **structurally minimal**, **fully deterministic**, and tightly integrated with both the vtable and the exception handling system. GCC relies on these encodings for safe polymorphic operations, cross-module type comparisons, and dynamic dispatch under arbitrary inheritance complexity—all without requiring runtime metadata lookup tables or language-level reflection infrastructure.

11.5 Examples: VTable Reverse Reconstruction from Binary

Reverse reconstruction of vtables from an ELF binary provides a direct, layout-accurate view of a class's runtime polymorphic structure. Because the **Itanium C++ ABI mandates a canonical vtable format**, the contents and ordering of the table can be inferred systematically from the binary without symbol-level recovery or debug information. This procedure is fundamental for binary compatibility audits, ABI regression analysis, decompilation, static security review, and reverse engineering of proprietary components.

This section demonstrates a methodical reconstruction of class hierarchy properties by inspecting `.rodata`, `.data.rel.ro`, and relocation table entries in an optimized, stripped executable.

11.5.1 Sample Class Hierarchy (Source)

```
struct A {
    virtual ~A() {}
    virtual void f();
};

struct B : virtual A {
    virtual void g();
};

struct C : A {
    virtual void h();
};

struct D : B, C {
```

```
    void f() override;
};
```

This hierarchy involves both **virtual inheritance** (`B` → `A`) and **multiple inheritance** (`D : B, C`). The Itanium ABI will generate multiple vtables and virtual base offset entries for `D`.

11.5.2 Identifying VTable Regions in the Binary

The vtables are emitted into **read-only relocation-adjusted tables**, typically under:

```
.data.rel.ro
.data.rel.ro.local
```

Search pattern:

1. Locate references to typeinfo objects:
   Typeinfo symbols follow the naming convention:

   ```
   _ZTI<encoded-name>
   ```

2. Scan for nearby objects prefixed with:

   ```
   _ZTV<encoded-name>
   ```

3. Confirm by checking that the table begins with:

   ```
   [ -2 ] RTTI pointer
   [ -1 ] offset-to-top
   [  0 ] function pointer entries
   ```

Example (annotated pseudo-disassembly):

```
_ZTV1D:
    00: 0x00000000004020a0   ; RTTI for D
    08: 0xfffffffffffffff0   ; offset-to-top = -16
    16: 0x0000000000401130   ; D::f()
    24: 0x00000000004010a0   ; inherited A::~A()
    32: 0x0000000000401090   ; inherited B::g() (via virtual base adjustment thunk)
    40: 0x0000000000401080   ; inherited C::h()
```

Values vary; structure remains constant.

11.5.3 Detecting Virtual Base Inheritance

Virtual base presence is encoded not in the vtable itself, but in the **typeinfo** object associated with the vtable. The `__vmi_class_type_info` structure includes the `offset_flags` array describing:

- Virtual inheritance markers
- Offset displacements to shared bases

You confirm virtual inheritance by inspecting the typeinfo graph:

```
typeinfo for D → vmi_class_type_info
  base_count = 2
  base[0] = B (non-virtual)
  base[1] = A (virtual)
```

This indicates:

- A single instance of `A` is shared across the entire `D` object.
- Some entries in the vtable for `D` will correspond to **adjustor thunks** to restore correct base pointer positions.

11.5.4 Recognizing Thunks in Reconstructed Dispatch Table

A thunk is identified by:

- A short function whose body performs pointer arithmetic on `rdi` (the `this` pointer)
- A tail call (`jmp`) to the actual implementation

Example assembly fragment observed during reconstruction:

```
thunk_B_g_for_D:
    add     rdi, -0x10      ; adjust this pointer to mapped B subobject
    jmp     0x401090 <B::g()>
```

This confirms:

- The vtable entry does **not** directly store the real method pointer.
- The ABI preserves correct dynamic dispatch semantics by pointer adjustment.

11.5.5 Reverse Inferring Class Relationship Structure

From the recovered vtable:

1. **Count the contiguous region of virtual function entries** → determines interface surface.
2. **Different offset-to-top values** across vtable variants → indicates number of inherited subobjects.
3. **Presence of adjustor thunks** → indicates non-trivial inheritance path requiring pointer correction.

4. **Typeinfo chain inspection** → reconstructs inheritance graph reliably.

The reversed class hierarchy obtained from the vtable and RTTI metadata (without source code) matches the original:

```
      A
     / \
virtual  \
    B     C
     \   /
       D
```

This is a structural property, not metadata or debug-data dependent.

11.5.6 Summary

Feature Observed	Interpretation
RTTI pointer at `vtable[-2]`	Identifies dynamic type root
Offset-to-top at `vtable[-1]`	Recovers correct `this` pointer for calls
Thunk entries	Adjust `this` for multiple or virtual inheritance dispatch
Multiple vtable sections for the same class	Indicates distinct subobject layout contexts
Typeinfo graph relationships	Reconstruct entire class inheritance topology

Reverse vtable reconstruction is reliable and deterministic because the Itanium ABI specifies a **fixed and observable runtime object representation**. GCC follows this definition rigorously, allowing the **object model** to be recovered from the binary with no source-level visibility, symbol names, or debug information.

Chapter 12

glibc Runtime, Static Initialization, and TLS Models

12.1 Startup Code (`crt1`, `crti`, `crtn`) and `_start` Transition

The GNU C Library (glibc) and GCC runtime architecture for ELF-based Linux systems defines a deterministic sequence of initialization and transfer stages between the kernel, the startup runtime (CRT objects), and the C++ application entry point (`main`). On x86-64, this transition is implemented through a precisely defined execution chain beginning at the kernel's `execve()` system call and ending in user code after runtime construction of the execution environment.

This sequence is controlled by `crt1.o`, `crti.o`, and `crtn.o`, which form the foundational **C runtime (CRT)** components that GCC links automatically before and after user object files.

12.1.1 Entry: Kernel to User Mode Transition

When a Linux ELF binary is executed, the kernel sets up:

- The initial stack containing `argc`, `argv`, and `envp`.
- The auxiliary vector (`auxv`) entries providing runtime parameters (page size, platform string, program header address, etc.).
- The entry instruction pointer (`_start`) pointing to the process's initial instruction in `crt1.o`.

This entry point bypasses glibc entirely at first. The CPU's state on entry is defined by the **System V AMD64 ABI**, not by the C or C++ runtime environment.
Registers upon entry:

```
RSP → argc
[RSP+8] → argv[0]
[RSP+(argc+1)*8] → envp[0]
```

No stack alignment, heap, or TLS setup beyond kernel guarantees is present yet.

12.1.2 `_start` Symbol in `crt1.o`

The `_start` symbol defines the *first instruction executed in user space*. It performs the following sequence:

1. Extracts `argc`, `argv`, and `envp` from the process stack.
2. Aligns the stack according to ABI (16-byte alignment).
3. Initializes the frame pointer and clears callee-saved registers.

4. Calls the internal symbol `__libc_start_main()` with:

    - The address of `main`
    - The addresses of constructors (`__libc_csu_init`)
    - The addresses of destructors (`__libc_csu_fini`)
    - Stack arguments and environment vectors

Conceptual flow:

```
_start:
    mov rdi, argc
    lea rsi, [rsp+8]           ; argv
    lea rdx, [rsp + (argc+2)*8]; envp
    call __libc_start_main
    hlt                        ; should never return
```

This defines the fixed point where **user-defined logic becomes reachable**.

12.1.3 `crti.o` and `crtn.o`: Constructor Frame Wrappers

`crti.o` and `crtn.o` are small object files that define **prologue** and **epilogue** sections used to construct global initialization routines at link time.
They provide stub definitions for `.init` and `.fini` sections:

- `crti.o` — introduces section prologue symbols (`_init` / `_fini` entry labels).
- `crtn.o` — closes the sections, completing the control flow frame.

When GCC compiles a translation unit containing global constructors or static objects, it emits **`.init_array`** entries. During final link:

```
[ crti.o prologue ]
[ object init code (.init_array) ]
[ crtn.o epilogue ]
```

The linker concatenates these into a single composite `_init` and `_fini` section that runs before and after `main()`.

12.1.4 `__libc_start_main()` Coordination

`__libc_start_main()` (from glibc) is responsible for:

1. Initializing thread-local storage (TLS) and dynamic linker state.
2. Executing pre-initialization functions (`.preinit_array`).
3. Calling global constructors (`.init_array` and `_init`).
4. Invoking the user `main(argc, argv, envp)`.
5. Calling destructors (`.fini_array`, `_fini`) upon exit.

This function provides the **boundary contract between CRT startup and the glibc runtime**.
The kernel never interacts directly with C++ initializers — all such activity is mediated by `__libc_start_main()`.

12.1.5 Static vs. Dynamic Linking Behavior

- **Static binaries**: All startup routines (`crt1.o`, `crti.o`, `crtn.o`, libc, and libstdc++ runtime stubs) are linked into one ELF segment. The entry point remains `_start`.

- **Dynamically linked binaries**: The dynamic linker (`ld-linux-x86-64.so.2`) is first mapped by the kernel as the interpreter (`PT_INTERP` segment). The dynamic linker's `_start` executes first, relocates shared libraries, and then transfers control to the application's `_start` defined in `crt1.o`.

The execution order for dynamic binaries:

```
Kernel → ld-linux → app's crt1::_start → __libc_start_main() → main()
```

12.1.6 Observing `_start` and CRT Symbols

To inspect startup symbols and section linkage in a binary:

```
$ readelf -s a.out | grep 'crt\|_start'
  0000000000401000  0 FUNC    GLOBAL DEFAULT 1 _start
  ...
$ objdump -d a.out | grep _start -A20
```

And to view the constructed initialization arrays:

```
$ readelf -r a.out | grep init_array
```

12.1.7 Summary

Stage	Component	Function
Process entry	Kernel → `_start` (crt1)	Initializes `argc`/`argv`/`envp` stack
Runtime prologue	`crti.o`	Creates `.init` and `.fini` section headers

Stage	Component	Function
Library entry	`__libc_start_main()`	Constructs TLS and executes global constructors
User execution	`main()`	Application-defined logic
Termination	`crtn.o` + glibc	Calls destructors and performs runtime cleanup

The **`crt1`–`crti`–`crtn` chain** forms the *compiler-agnostic mechanical base* of all C++ binaries under Linux, linking kernel state to user logic in a reproducible and ABI-stable way. Every GCC-compiled program, whether static or dynamically linked, passes through this precisely defined transition pipeline before invoking any C++ runtime functionality.

12.2 TLS Model Selection (local-exec, initial-exec, local-dynamic)

Thread-Local Storage (TLS) allows each thread to maintain private instances of global or static variables. The ELF and glibc runtime define multiple **TLS access models**, each representing a trade-off between **performance**, **relocation flexibility**, and **dynamic linking compatibility**. GCC selects a TLS model during compilation based on symbol visibility, linkage, and optimization assumptions. The correct TLS model choice is critical for both ABI stability and execution efficiency on Linux x86-64.

12.2.1 TLS Access Models in the Itanium ABI

Four models are relevant in ELF systems:

Model	Requires Position?	Can Be Used in Shared Libraries?	Performance Characteristics
local-exec	Non-position-independent (no relocations)	No	Fastest; direct access via `%fs` segment
initial-exec	PIC allowed but variable must be non-interposable	Yes, if symbol resolves at load time	Fast; single GOT lookup
local-dynamic	Fully position independent	Yes	Runtime TLS block offset resolution per module

Model	Requires Position?	Can Be Used in Shared Libraries?	Performance Characteristics
general-dynamic	Fully general resolution	Yes	Slowest; full dynamic TLS lookup sequence

The **compiler**, **linker**, and **dynamic loader** cooperate to resolve which model is valid in a given context.

12.2.2 Segment Register and TLS Memory Layout

On x86-64 Linux, each thread has a **Thread Control Block (TCB)** referenced by the `%fs` register. TLS variables are stored at fixed offsets relative to the TCB inside a per-thread memory region allocated at thread creation.
Accessing a TLS variable in the **local-exec** model is simply:

```
mov rax, qword ptr fs:offset
```

This requires **no GOT indirection** and offers **register-level latency**, making it the preferred model for performance-critical code such as allocators, memory pools, or hot-path schedulers.

12.2.3 Local-Exec Model

Used when:

- The binary is statically linked (`-static`), or
- The TLS variable is **hidden visibility** and known to reside in the main module.

Example:

```
__attribute__((tls_model("local-exec")))
thread_local int counter;
```

This instructs GCC to emit direct `%fs`-relative access sequences, minimizing runtime overhead. However, this model **cannot** be used if the variable may reside in a shared library loaded at runtime.

12.2.4 Initial-Exec Model

Used when the variable is:

- In a shared library **but not interposable** (typically `STV_HIDDEN` or `STV_PROTECTED`).
- Known to be bound at program load time (no `dlopen` relocation).

Access form:

```
mov rax, qword ptr [GOT entry]
mov rax, fs:[rax]
```

This requires a fixed GOT offset that is resolved only once during program load. The cost is low: **one memory indirection + segment-base load**. This is the most common model for TLS when using shared libraries with standard C++ runtimes.

12.2.5 Local-Dynamic Model

Used when PIC is required and the TLS block offset must be resolved at runtime, but the variable's module is known.
TLS access sequence:

```
call __tls_get_addr@plt
mov rax, [rax + var_offset]
```

The call resolves the thread-specific offset **for the module**. Cost: function call + arithmetic. This model trades flexibility for moderate overhead.

12.2.6 General-Dynamic Model

The fallback model used when:

- The symbol may be interposable across shared libraries.
- The compiler cannot assume early binding.

Access requires:

```
call __tls_get_addr@plt    ; resolves module + symbol offset
```

Used when no compile-time assumptions are valid.
This model is **significantly slower** and should be avoided in hot paths.

12.2.7 Compiler and Linker Selection Rules

GCC applies the following heuristics:

Symbol Property	Selected TLS Model
`thread_local` with hidden visibility in main executable	**local-exec**
`thread_local` in shared library with default visibility	**local-dynamic** or **general-dynamic**

Symbol Property	Selected TLS Model
`-fno-pic` or static linking	**local-exec**
Link-time optimization proving symbol non-interposable	**initial-exec**

User override:

```
__attribute__((tls_model("initial-exec")))
thread_local int flag;
```

12.2.8 Summary

Model	Pros	Cons	Best Use
local-exec	Fastest TLS access	Not usable in shared libraries	Performance-critical static binaries
initial-exec	Fast access with PIC	Requires non-interposition guarantees	Shared libraries with fixed runtime binding
local-dynamic	Flexible module TLS offsets	Requires `__tls_get_addr` call	General shared library TLS with moderate cost
general-dynamic	Fully interposable	Slowest TLS model	`dlopen`-loaded or ABI-unconstrained libraries

TLS model selection is therefore a **link-time ABI decision**, not a purely compiler-local choice. Correct use ensures both **high-performance TLS access** and **runtime relocation correctness across shared object boundaries**.

12.3 Constructor Order Resolution and Guard Variable Semantics

C++ relies on **deterministic construction of global and static-duration objects** before entering `main()`, and deterministic destruction after `main()` completes. The glibc runtime and GCC cooperate to sequence initialization operations through the `.init_array`, `.preinit_array`, and `.fini_array` tables, while enforcing **one-time initialization semantics** for local static objects using **guard variables**. This section describes the ordering, enforcement rules, and generated machine code sequences associated with these mechanisms.

12.3.1 Global and Namespace-Scope Static Initialization

For each translation unit, GCC emits constructors into the `.init_array` segment. At runtime, during `__libc_start_main()`:

1. `.preinit_array` functions run (rarely used; reserved for runtime frameworks).
2. `.init_array` constructors run in the exact order they appear in the final link.
3. `main()` is invoked.

Order is **link-order, not source-order**, which means:

- Relative constructor order **across translation units** is unspecified unless controlled via link script order or explicit initialization dependencies.
- Within a single translation unit, with standard global variable declarations, **top-to-bottom declaration order is preserved**.

This ordering cannot be used to encode runtime dependencies unless explicitly documented.

12.3.2 Dynamic Initialization vs. Static Initialization

Two initialization categories exist:

Initialization Type	Timing	Example	Cost
Static	Performed at load time; generates relocations only	`constexpr int x = 42;`	No runtime overhead
Dynamic	Executed via `.init_array` at runtime	`std::string s = "hello";`	Runs code before `main()`

GCC places dynamic initialization code into synthetic functions referenced from `.init_array`. The code runs sequentially in monothreaded context early in process startup, avoiding race conditions by construction.

12.3.3 Local Static Initialization and Guard Variables

Local static objects must follow the **one-time initialization rule** mandated by the C++ standard:

```
void f() {
    static Widget w; // must initialize exactly once, even under concurrency
}
```

To enforce this, GCC emits a **guard variable** and lock-free test mechanism around the initialization block:

```
    mov    al, BYTE PTR guard_variable[rip]
```

```
    test   al, al
    jne    .Linit_done
    call   __cxa_guard_acquire(guard_variable)
    ; construct w
    call   __cxa_guard_release(guard_variable)
.Linit_done:
```

If initialization fails (exception thrown):

```
call __cxa_guard_abort(guard_variable)
```

Guard variables are encoded with type `__guard` ABI rules:

- **1-byte fast check** for initialization completion
- Atomic acquire/release semantics when multi-threaded support is linked (`-pthread` or libstdc++ with concurrency support)
- No OS locks unless required by preemption context

This ensures correctness under concurrent execution even when multiple threads first call the same function.

12.3.4 Interaction with TLS (`thread_local` Objects)

`thread_local` variables follow the same one-time initialization semantics, but **per-thread**.
Their guards are stored in the thread's TLS block, ensuring no cross-thread interference.
Example:

```
void f() {
    thread_local std::vector<int> q;
}
```

Initialization must occur once **per thread**, not globally. The guard variable lives in TLS and uses the same `__cxa_guard_*` logic but scoped to thread lifetime.

12.3.5 Destructor Ordering and Program Shutdown

Destructors for static-duration objects are registered via:

```
__cxa_atexit(function_pointer, object_pointer, dso_handle)
```

The `dso_handle` uniquely identifies each shared object. Destructors run in **reverse construction order**, obeying dependency consistency.
Order of teardown:

1. Objects in the executable's `.fini_array`
2. Objects in shared libraries, in reverse order of loading
3. TLS destructors for each thread at thread exit

This ensures no object is destroyed before another that may depend on it.

12.3.6 Summary

Mechanism	Purpose	Runtime Effect
`.init_array`	Sequencing of global constructors	Deterministic initialization before `main()`
Guard variable	Enforce one-time static initialization	Thread-safe lazy initialization

Mechanism	Purpose	Runtime Effect
`__cxa_guard_{acquire,release,abort}`	Implements guard semantics	Prevents races and double initialization
`.fini_array` + `__cxa_atexit`	Reversed destruction ordering	Ensures dependency-safe resource teardown
TLS guard storage	Per-thread static initialization	No cross-thread interference for `thread_local`

The constructor and guard variable mechanisms form a core part of the **C++ runtime execution contract**: initialization of global and local static objects is guaranteed to be **correct, ordered, and thread-safe**, while still supporting zero-cost access after the initial setup. GCC and glibc implement this behavior in a manner that is stable across shared libraries, dynamic loading boundaries, and multithreaded environments.

12.4 Shutdown Ordering and Finalization Guarantees

The C++ runtime provides **deterministic guarantees** for the destruction of static-duration and thread-local objects at program and thread termination. These guarantees are required for correctness in resource management, especially when involving file handles, memory allocators, mutexes, and user-defined RAII abstractions. On Linux x86-64, finalization is orchestrated jointly by **glibc**, **libstdc++**, and the **Itanium C++ ABI** functions that manage destructor registration and invocation.

12.4.1 Global Object Finalization via `__cxa_atexit`

Every dynamic initialization of a static-duration object registers a destructor with:

```
__cxa_atexit(void (*destructor)(void*), void* object, void* dso_handle);
```

Where:

- `destructor` is the function to be called at finalization,
- `object` is the instance to be destroyed,
- `dso_handle` identifies the shared object or executable in which the object resides.

Registration order follows **construction order**, guaranteeing **reverse-order destruction**, implementing a strict LIFO semantics globally.
This ensures that any object depending on another constructed earlier remains valid during its own finalization.

12.4.2 Shared Library Unloading and DSO Handles

In dynamically linked programs, shared libraries may be unloaded before program exit (e.g., after `dlclose`).
To maintain correctness:

- Each library receives its own `dso_handle`.
- Destructors registered from that library are grouped and executed when the library is unloaded.
- If the application terminates normally, destructors are run in the reverse order of library dependency loading.

This ensures correctness even in complex plugin architectures.

12.4.3 Finalization Ordering Across Translation Units

Although destructors run in reverse order of constructor calls, constructor order across translation units is not defined by the standard. The linker, not the compiler, determines the physical order of `.init_array` entries.
However, destruction still respects:

```
last constructed → first destroyed
```

This property allows stable resource dependency chains when initialization is explicitly structured (e.g., through factory or singleton patterns), while discouraging implicit cross-TU static initialization coupling.

12.4.4 Termination vs. Exit Path Semantics

Finalization only occurs under **regular termination paths**:

Termination Method	Are Global Destructors Run?	Notes
`return` from `main()`	Yes	Normal exit sequence
`std::exit()`	Yes	Calls `__cxa_atexit` destructors
`std::quick_exit()`	No	Calls only `at_quick_exit` handlers
`_Exit()` / `_exit()`	No	Immediate process termination
`abort()`	No	Abnormal termination; no unwinding

Production systems must ensure termination matches resource lifetime expectations.

12.4.5 Thread Exit and TLS Destructors

For objects declared `thread_local`:

- Destructors run at **thread termination**, not at program exit.
- glibc registers thread-specific destructor lists in **TLS control blocks**.
- When a thread exits (via `pthread_exit`, thread returning from start function, or cancellation), destructors run in reverse initialization order for that thread only.

This ensures:

```
Per-thread resources are reclaimed deterministically.
```

TLS destructors must not assume global objects still exist if threads persist past main thread exit.

12.4.6 Shutdown Ordering Example

```
struct Logger {
    Logger()  { /* open file */ }
    ~Logger() { /* flush, close file */ }
};

thread_local Logger local_log;
static Logger global_log;

int main() {
    // work
}
```

Shutdown sequence:

1. `main()` returns.
2. Global destructors run in reverse construction order (`~Logger()` for `global_log`).
3. If any threads remain:
    - Their TLS destructors run when each thread exits.
4. glibc final process cleanup occurs last.

If threads continue running after `main()` exits, TLS destructors for those threads will run *after* global static destructors, implying resource dependency reversal is possible and must be avoided in design.

12.4.7 Summary

Mechanism	Ordering Rule	Scope	Notes
`.init_array` / `.fini_array`	Reverse of construction order	Process-wide	Core global object lifetime management
`__cxa_atexit` registry	LIFO destruction	Per shared object	Ensures correctness across dynamic loading
TLS destructor chains	Reverse of per-thread construction order	Per thread	Runs at thread exit, not process exit
Normal vs. abnormal termination handling	Deterministic vs. skipped finalization	Whole process	Affects cleanup correctness and resource guarantees

Finalization ordering is a strictly defined and ABI-stable component of C++ object lifetime semantics. GCC and glibc ensure that all static and thread-local destructors run predictably **only in well-defined exit paths**, preserving invariants required for safe RAII-based resource management.

12.5 Examples: Instrumenting Global Initialization Graphs

Static and global object initialization forms an implicit dependency graph across translation units, shared libraries, and the C++ runtime. Correctness and performance often depend on understanding this graph—particularly when dealing with subsystems such as logging frameworks, memory allocators, or device interfaces that must be available before use. This section demonstrates **practical methods to trace, visualize, and reason about global initialization sequences** in a GCC + glibc runtime environment on Linux x86-64.

The objective is not to avoid global objects entirely, but to **make their initialization and finalization behavior explicit, observable, and verifiable**.

12.5.1 Basic Instrumentation via Constructor Attributes

GCC supports function attributes that allow attaching custom initialization routines into `.init_array`:

```
#include <cstdio>

__attribute__((constructor))
static void init_A() {
    std::puts("init_A");
}

__attribute__((constructor))
static void init_B() {
    std::puts("init_B");
}
```

```
int main() {
    std::puts("main");
}
```

Running this binary yields a runtime ordering trace:

```
init_A
init_B
main
```

This provides **coarse-grained ordering**, but does not show per-object construction.

12.5.2 Instrumenting Individual Static Objects

To observe per-object initialization, wrap global instances with logging behavior:

```
struct Trace {
    const char* name;
    Trace(const char* n) : name(n) { std::printf("Construct: %s\n", name); }
    ~Trace() { std::printf("Destruct: %s\n", name); }
};

static Trace A("A");
static Trace B("B");
```

Execution:

```
Construct: A
Construct: B
main
Destruct: B
Destruct: A
```

This confirms **reverse-order destruction** and link-order construction.

12.5.3 Detecting Cross-Translation-Unit Initialization Dependencies

Consider two translation units:

file1.cpp

```
extern int init_B();

int init_A() {
    return init_B() + 1; // dependent on B
}

static int A = init_A();
```

file2.cpp

```
int init_B() { return 10; }
static int B = init_B();
```

Compile and inspect constructor ordering:

```
g++ file1.cpp file2.cpp -o app -Wl,--no-as-needed -Wl,--verbose 2>&1 | grep init_array
```

If `file1.o` appears before `file2.o`, `init_A()` executes before `init_B()`, violating assumed dependency ordering.
There is **no standard-guaranteed sequencing** across translation units.
Mitigation requires either:

- converting to runtime initialization control (explicit initialization function),
- using function-local statics with guard variables,
- or consolidating static dependencies into single compilation units.

12.5.4 Visualizing `.init_array` Contents

Use `readelf` and `objdump`:

```
$ readelf -a app | grep init_array
```

Then inspect referenced constructor functions:

```
$ objdump -d --section=.init_array app
```

Each entry typically holds a pointer to compiler-synthesized initialization functions such as `_GLOBAL__sub_I_<symbol>`.

12.5.5 Full Initialization Graph Extraction

The following script extracts and annotates initialization function call traces:

```
objdump -d app |
  awk '/_GLOBAL__sub_I_/ {print $1}' |
  while read sym; do
      echo "Constructor: $sym"
      objdump -d --disassemble=$sym app | sed 's/^/  /'
  done
```

This allows:

- Mapping which static objects originate from which translation units.
- Detecting ordering cycles and unintended dependencies.
- Verifying initialization transitivity guarantees.

12.5.6 Runtime Graph Representation

A recommended representation of initialization ordering is a **directed acyclic graph (DAG)** where:

- Nodes = static or thread-local objects
- Edges = "must-be-initialized-before" relationships

Cycles imply invalid hidden dependencies and require architectural restructuring.

12.5.7 Summary

Task	Technique	Key Insight
Observe constructor order	`__attribute__((constructor))` or tracing class constructors	Confirms link-order sequencing
Trace per-object initialization	Logging constructors/destructors	Reveals hidden inter-object dependencies
Detect cross-TU dependency hazards	Compare `.init_array` and symbol visibility	Ordering across translation units is not guaranteed
Visualize initialization DAG	Extract `_GLOBAL__sub_I_` symbols	Enables structural correctness validation

Instrumenting global initialization is essential in advanced C++ system design because it transforms **implicit object lifetime contracts** into **explicit, analyzable behavior**.

This supports stable runtime architecture, reduces startup nondeterminism, and prevents subtle bugs related to uninitialized subsystems or resource lifetimes.

Chapter 13

Memory Allocation Internals and Latency Control

13.1 ptmalloc Arena Design and Cache Locality

The GNU C Library allocator, **ptmalloc2**, provides the default implementation of `malloc`, `free`, and related allocation services on Linux x86-64. Its design is based on dlmalloc but extended to support **multi-threaded scalability** through the concept of **arenas**. Each arena contains metadata and data regions used to satisfy allocations, allowing threads to reduce contention by operating mostly within their **assigned arena**. For performance-critical C++ systems, understanding how arenas influence locality, fragmentation, and load distribution is essential for predictable allocation latency.

13.1.1 Arena Structure Overview

An **arena** consists of:

- A **top chunk** representing the current expandable end of the heap.
- One or more **bins**, each holding free blocks grouped by size class.
- A **thread ownership model**, where a thread may acquire an arena and reuse it to reduce lock contention.

Memory belonging to an arena is typically obtained via:

1. `mmap` (for large allocations or multiple arenas), or
2. `sbrk` (primary heap expansion for the main arena).

Each arena serves allocation requests **locally** without requiring coordination with others, unless the requested block cannot be satisfied within its own bin set.

13.1.2 Multi-Arena Behavior and Thread Locality

By default, glibc selects the arena based on:

- The calling thread.
- Whether the requested block size fits within per-size free lists.
- The number of existing arenas relative to the number of CPU cores.

A common heuristic is:

```
number_of_arenas  8 × number_of_CPU_cores
```

This ensures that **threads rarely contend** for the same arena lock.
Because each arena reuses blocks previously freed to it, **allocation and free operations exhibit locality over time**, particularly in thread-bound workloads. However, if threads migrate across cores or if tasks are scheduled non-deterministically, blocks may be allocated by one thread and freed by another, causing **cross-arena memory movement**, which reduces locality and may introduce fragmentation.

13.1.3 Cache Locality and Allocation Patterns

Cache locality is influenced by:

- **Temporal locality**: Recently freed blocks tend to be reused first.
- **Spatial locality**: Smaller bins store blocks close together in memory.
- **Arena affinity**: Threads often operate on the same arena repeatedly, reinforcing locality.

However, large allocations trigger `mmap`, which:

- Allocates dedicated regions separate from arenas.
- Keeps overhead low but increases fragmentation risk for irregular workloads.

For performance-critical loops that allocate and free frequently, using custom allocators or scoped storage (`std::pmr::monotonic_buffer_resource`) can dramatically reduce cache misses and synchronization overhead.

13.1.4 Binning and Coalescing Strategy

ptmalloc uses **segregated free lists** ("bins") categorized by block sizes. Small allocations (e.g., < 512 bytes) are handled from dedicated small bins optimized for locality, while larger allocations use tree bins that trade locality for flexible fit selection. Free blocks adjacent in memory are **coalesced** automatically, reducing fragmentation but requiring conditional metadata checks. These coalescing operations are **constant-time** due to the chunk header linkage structure, but may cost CPU cycles under heavy multi-threaded churn.

13.1.5 Impact on C++ Allocator Behavior

C++ abstractions are layered on top of ptmalloc:

Allocation Layer	Mechanism	Notes
`operator new/delete`	Calls `malloc/free`	Performance inherits `ptmalloc` arena behavior
`std::allocator<T>`	Thin wrapper	Does not guarantee locality across containers
`pmr` memory resources	Optional custom arenas	Can enforce locality and eliminate contention

Performance-sensitive components (e.g., message queues, lock-free structures, schedulers) benefit from:

- **per-thread memory pools**,
- **slab allocators**, or
- **monotonic region allocators**.

These avoid the unpredictability of arena switching and lock acquisition.

13.1.6 Practical Diagnosis

To analyze arena behavior:

```
$ MALLOC_ARENA_MAX=1 ./app
```

Reduces number of arenas for repeatable tracing.

```
$ LD_PRELOAD=/usr/lib/libtcmalloc.so ./app
```

Substitutes allocator for comparison testing.
To inspect chunk layout:

```
$ gdb -ex "set print pretty on" -ex "call malloc_stats()" --args ./app
```

13.1.7 Summary

Property	Effect on Performance
Per-thread arenas	Reduces lock contention; improves scalability
Bin-based size classing	Improves reuse locality; predictable small-block allocation
Coalescing + metadata overhead	Controls fragmentation at moderate CPU cost
Large allocations use `mmap`	Reduces overhead but may fragment the address space
Thread migration	May reduce arena locality and induce cross-core traffic

ptmalloc's design is optimized for **general-purpose robustness**, not for minimal-latency deterministic allocation. Performance-critical C++ architectures should consider **explicit memory locality management**, either through standard polymorphic memory resources or custom domain-specific allocators.

13.2 Multithreaded Allocator Contention and Arena Replication

In multi-threaded C++ applications, memory allocation frequently becomes a shared synchronization point. The default glibc allocator, **ptmalloc2**, mitigates contention using **multiple arenas**, allowing concurrent threads to allocate memory independently. However, arena replication introduces trade-offs in locality, fragmentation, and scalability. Understanding these behaviors is necessary to reason about allocator-induced latency under real workloads.

13.2.1 Arena Acquisition and Thread Mapping

When a thread requests memory, ptmalloc attempts to assign it to an existing arena. If an arena is free (its lock is not held), the thread acquires it and continues allocating exclusively within it. If multiple threads concurrently attempt to acquire the same arena, lock contention occurs.
If all arenas are busy, and the number of arenas has not reached the system's allowed threshold:

```
max_arenas  8 × number_of_CPU_cores
```

then ptmalloc **creates a new arena** via `mmap`. The newly created arena is then assigned to the contending thread.
This mechanism spreads allocator load across CPU cores and reduces the likelihood of lock contention but increases:

- **Total resident memory footprint**
- **Fragmentation across arenas**
- **Cross-thread allocation ownership inconsistencies**

13.2.2 Arena Locking Granularity and Fast Path Behavior

Each arena has a central **mutex** protecting:

- Bin lookup and updates
- Coalescing operations
- Top chunk extension

Small allocations are served aggressively from size-class bins; if the bin contains suitable free blocks, allocation resolves with:

```
lock → remove chunk → unlock
```

The lock duration is short but still serialized. If the thread repeatedly uses the same arena, locality is preserved and cache reuse remains high. When a thread switches arenas, two penalties occur:

1. **Allocator lock migration** across CPU cores.
2. **Reusable free blocks** belonging to different memory regions, reducing cache locality.

Threads migrating frequently between arenas are characteristic of systems with dynamic scheduling or work-stealing task runtimes.

13.2.3 Fragmentation from Cross-Arena Freeing

When memory allocated in one arena is freed by a thread associated with a different arena, ptmalloc must:

- Acquire the arena lock of the block's originating arena.

- Return the block to the correct bin and possibly coalesce it.

This results in:

- **Allocator lock traffic between threads**
- **Poor temporal locality**, since memory freed may not be reused by the freeing thread
- **Increased fragmentation** when allocation/free patterns become non-local

This behavior is particularly costly in producer/consumer systems where ownership transfer is high.

13.2.4 NUMA Effects and Core Affinity

On NUMA systems, arena replication interacts with memory locality:

Behavior	Result
Thread repeatedly allocates/free in the same arena	Memory resides in local NUMA node; cache locality maintained
Thread migrates cores	Accesses remote-memory arenas; latency increases
Thread frees memory allocated by another NUMA domain	Inter-node traffic and increased LLC pressure

This makes allocator behavior **latency-sensitive to scheduler decisions**.
For systems requiring strict memory locality (e.g., real-time engines, HPC kernels), binding threads to cores (`sched_setaffinity`) significantly improves allocator performance.

13.2.5 Contention Diagnostics

To diagnose arena contention:

```
$ MALLOC_ARENA_MAX=1 ./app
```

Forces single-arena mode, exposing lock stalls directly.
Use `perf` to measure lock wait times:

```
$ perf lock record ./app
$ perf lock report
```

High lock wait indicates allocator pressure; solutions include:

- Thread-local pooling (`thread_local` slab or object pools)
- PMR monotonic or fixed-size buffer resources
- Choosing an alternative allocator (e.g., tcmalloc, jemalloc, mimalloc)

13.2.6 Summary

Mechanism	Benefit	Cost
Arena replication	Reduces locking contention	Increases fragmentation and memory footprint
Per-arena free lists	High locality when threads do not migrate	Locality collapses under thread movement
Cross-arena free handling	Correct memory return	Introduces inter-thread lock traffic

Mechanism	Benefit	Cost
Core and NUMA affinity	Stabilizes allocator locality	Requires scheduler-aware application design

Arena replication enables **scalable average-case performance**, but only when **thread locality is stable**. Systems that move work dynamically across threads will incur allocator-induced latency unless memory allocation is explicitly designed to be **thread-local or region-based**.

13.3 Custom Allocators for STL Containers

The default allocator used by standard containers (`std::allocator<T>`) delegates to `operator new`, which in turn relies on the global heap allocator (ptmalloc2 in glibc). While this is sufficient for general-purpose workloads, performance-critical and concurrent systems benefit from **custom allocator strategies** that reduce contention, improve cache locality, and control object placement. The C++ allocator model enables replacing the default allocator with container-aware and domain-specific allocation behavior.

13.3.1 Allocator Model Requirements

C++20 defines the allocator interface through `std::allocator_traits`, separating allocator *policy* from allocator *binding*. Any custom allocator must define:

```
template<class T>
struct Allocator {
    using value_type = T;

    T* allocate(std::size_t n);
    void deallocate(T* p, std::size_t n) noexcept;
};
```

All higher-level semantics—object construction, destruction, rebind, propagation on move—are derived automatically through `std::allocator_traits`.
This separation allows:

- Allocators to be shared across container instances,
- Per-container memory pools,
- Region or arena-backed allocation.

13.3.2 Motivations for Custom Allocators in High-Performance Systems

Requirement	Default Allocator Behavior	Custom Allocator Advantage
Cache locality	Objects may be widely dispersed	Allocator may enforce contiguous block placement
Multithreaded scalability	Dependent on arena behavior	Thread-local pools eliminate lock contention
Deterministic latency	Allocation requires metadata lookup	Fixed-size pools provide O(1) allocation
Memory ownership control	Lifetime is implicit	Region/arena destruction is constant time

In performance-critical systems, **memory locality and synchronization behavior** matter more than raw allocation throughput.

13.3.3 Pool Allocators for Fixed-Size Objects

For containers such as `std::vector<T>` or `std::deque<T>`, object size is known and stable. A **slab or pool allocator** pre-allocates a large contiguous region and serves objects from it:

```
template<class T>
class PoolAllocator {
public:
    using value_type = T;
```

```
    T* allocate(std::size_t n) {
        return static_cast<T*>(pool.allocate(n * sizeof(T)));
    }

    void deallocate(T* p, std::size_t) noexcept {
        pool.deallocate(p);
    }

private:
    ThreadLocalPool pool;
};
```

Benefits:

- Minimal fragmentation
- Cache-coherent iteration
- No interaction with the global heap

13.3.4 Monotonic and Region-Based Allocation

C++17 introduced `std::pmr` to simplify allocator design. `std::pmr::monotonic_buffer_resource` allocates memory in **growing regions**, with no individual deallocation:

```
std::pmr::monotonic_buffer_resource buffer;
std::pmr::vector<int> v(&buffer);
```

Advantages:

- Zero per-object free cost

- Lifetime tied to region scope
- Excellent locality for construction-heavy phases

This model is ideal for parse trees, compiler front-ends, scene graphs, and batch computations.

13.3.5 Thread-Local Allocators for Concurrency

To avoid arena contention, a **thread-local allocator** isolates allocation to the executing thread:

```
thread_local std::pmr::monotonic_buffer_resource thread_pool;
std::pmr::unordered_map<Key, Value> map(&thread_pool);
```

This eliminates allocator locks entirely so long as objects remain local to the thread. When cross-thread ownership transfer is required, memory ownership must be explicitly mediated—either via shared-memory transfer queues or per-thread reclamation systems such as hazard-pointer based reclamation.

13.3.6 Performance Considerations and Trade-offs

Allocator Type	Strengths	Limitations
Pool / Slab	Consistent latency; strong locality	Must manage type sizes explicitly
Monotonic Region	Fast allocation and teardown	No granular deallocation; must reset region at controlled points

Allocator Type	Strengths	Limitations
Thread-Local	Zero contention	Requires strict ownership discipline
Global Arena-Backed (`std::allocator`)	Simple; universal semantics	Potential contention and reduced locality under concurrency

13.3.7 Summary

Custom allocators transform the memory allocation layer from a **general-purpose heap** into a **domain-optimized storage strategy**. In systems where memory allocation is on the critical path—such as schedulers, real-time control loops, messaging fabrics, indexing engines, or UI event pipelines—allocator selection directly influences:

- Cache locality and coherence behavior
- Tail latency under contention
- Memory footprint stability
- Overall determinism of execution time

The allocator is therefore not a peripheral implementation detail but a **first-order architectural parameter** in advanced C++ system design on Linux.

13.4 Using ASan + Heaptrack to Diagnose Fragmentation

Efficient memory allocation in complex C++ systems is not solely determined by raw allocator throughput; **fragmentation patterns, lifetime mismatches, and unintended allocation hot paths** often dominate performance behavior. To identify and correct such issues, modern toolchains provide two complementary diagnostic tools:

- **AddressSanitizer (ASan)** for detecting incorrect memory usage.
- **Heaptrack** for tracing allocation patterns, fragmentation, and allocator pressure.

Together, they form a **behavioral and structural analysis pipeline** that reveals the root causes of allocation-induced latency and memory growth.

13.4.1 Why ASan and Heaptrack Are Complementary

Tool	Primary Focus	Strengths	Limitations
ASan	Detect invalid memory behavior (use-after-free, buffer overflow)	Strong correctness enforcement; immediate failure reporting	Does not analyze fragmentation dynamics
Heaptrack	Observe allocation size, frequency, and temporal patterns	Reveals fragmentation and allocation hotspots	Does not detect invalid memory access

ASan addresses **safety**; Heaptrack addresses **efficiency**. Correctness and performance diagnostics must be performed jointly.

13.4.2 Building and Running with ASan

Enable ASan during compilation:

```
$ g++ -O2 -fsanitize=address -fno-omit-frame-pointer app.cpp -o app_asan
```

Key options:

- `-fsanitize=address` enables redzones and shadow memory.
- `-fno-omit-frame-pointer` allows accurate stack trace resolution.
- Avoid `-flto` unless using a version of LLVM with matching sanitizer runtime compatibility.

Running detects:

- Heap-use-after-free
- Stack and heap buffer overflows
- Double free
- Incorrect delete/delete[] mismatches

Example failure output:

```
==12345==ERROR: AddressSanitizer: heap-use-after-free ...
```

If ASan reports no violations, fragmentation and allocator inefficiency become the primary suspects.

13.4.3 Collecting Heaptrack Traces

Run the application under Heaptrack:

```
$ heaptrack ./app
```

This produces a `.gz` trace file. Analyze the trace:

```
$ heaptrack_gui heaptrack.<pid>.gz
```

Key signals to examine:

- **Hot allocation call stacks**: repeated allocation inside loops.
- **Lifetime mismatch chains**: long-lived objects retaining large blocks.
- **Cross-thread memory free patterns**: indicative of loss of arena locality.
- **Large `mmap` event frequency**: implies non-locality or allocator exhaustion.

Heaptrack's flame graphs reveal allocation "pressure zones" where redesign or custom allocators may be warranted.

13.4.4 Diagnosing Fragmentation Patterns

Common fragmentation indicators include:

Symptom	Root Cause	Mitigation
Many small freed blocks remain unused	Interleaving lifetimes in shared arena	Convert to monotonic region or pool allocator

Symptom	Root Cause	Mitigation
Frequent `mmap` / `munmap` cycles	Oversized allocations bypassing bins	Introduce slab partitioning for large objects
Memory footprint grows without leak	Long-lived containers holding many small nodes	Compact underlying data structures or use `reserve()`
Highly variable allocation latency	Cross-arena deallocation from thread migration	Introduce thread-local pools or affinity policies

Heaptrack provides context: which code path allocates, how frequently, and with which lifetime distribution.

13.4.5 Combining ASan and Heaptrack in Diagnostic Workflow

Typical workflow:

1. **Run with ASan**
   Ensure correctness.
   If errors are found, fix them before investigating performance.

2. **Run with Heaptrack**
   Capture full-system allocation behavior.

3. **Inspect Hot Paths**
   Identify functions responsible for allocation volume, churn, or waste.

4. **Redesign Allocation Strategy**
   Apply:

- `reserve()` for vectors,
- `shrink_to_fit()` when appropriate,
- `std::pmr` for region-based lifetimes,
- thread-local slab allocators for concurrency.

5. **Re-measure under realistic load**
   Verify behavior under concurrent stress.

13.4.6 Summary

Objective	Tool	Result
Ensure memory correctness	AddressSanitizer	Prevent corruption, use-after-free, overflow, and double free
Understand memory behavior	Heaptrack	Reveal allocation patterns, growth, and fragmentation
Improve stability and throughput	Custom allocator strategy	Reduce latency, contention, and memory footprint

This combined diagnostic approach converts allocator behavior from **implicit emergent patterns** into **observable, controllable system properties**.
In advanced C++ systems, memory behavior must be treated as a **first-class architectural dimension**, not an incidental runtime detail.

13.5 Examples: Optimizing Allocator for `std::vector` Reuse Patterns

`std::vector` is one of the most commonly used containers in performance-sensitive C++ systems. Its behavior is predictable: it stores elements contiguously in memory and grows its capacity geometrically (typically by a factor of 1.5 or 2). However, in systems where vectors are repeatedly created and destroyed—such as in frame-based simulation pipelines, request batching layers, or message processing loops—the default allocator may induce:

- Repeated heap calls for allocation and deallocation
- Loss of cache locality due to memory churn
- Increased pressure on the ptmalloc arena structure
- Latency spikes from top-chunk growth and coalescing

To optimize for such **reuse-heavy patterns**, we replace the default allocator with **monotonic**, **pool-backed**, or **small-buffer optimized** strategies that allow `std::vector` to reuse previously acquired memory rather than repeatedly interacting with the general heap.

13.5.1 The Problem: Transient Vectors in Tight Loops

Example pattern:

```
for (;;) {
    std::vector<float> data;      // allocate
    data.reserve(4096);           // heap expansion
    process(data);
}                                   // deallocate
```

This creates and frees a heap buffer *every iteration*, defeating cache locality and increasing allocator contention.
Even if the vector is outside the loop:

```
std::vector<float> data;
for (;;) {
    data.clear();   // does NOT free capacity
    process(data);
}
```

The underlying capacity may still be reallocated if the workload occasionally requires temporary growth beyond previously seen sizes.
The optimization goal is to **tie vector allocation lifetime to a reuse scope**.

13.5.2 Using `std::pmr::monotonic_buffer_resource`

```
#include <memory_resource>
#include <vector>

thread_local std::pmr::monotonic_buffer_resource buffer;

void run() {
    std::pmr::vector<float> v(&buffer);
    v.reserve(4096);
    process(v);
}
```

Properties:

Behavior	Effect
Allocations are taken from `buffer`	No calls to `malloc` during reuse
No per-element deallocation	Destruction is `O(1)` when buffer resets
Memory locality	Guaranteed for all vectors using the same region

Best for repeated bulk-compute phases with well-bounded vector sizes.

13.5.3 Pool Allocator for Stable Object Sizes

For vectors with **fixed or predictable maximum size**, use a **slab allocator** that supplies contiguous blocks:

```
template<class T, std::size_t Max>
class SlabAllocator {
public:
    using value_type = T;
    T* allocate(std::size_t n) {
        if (n > Max) throw std::bad_alloc();
        return reinterpret_cast<T*>(storage);
    }
    void deallocate(T*, std::size_t) noexcept {}
private:
    alignas(T) static inline std::byte storage[sizeof(T) * Max];
};
```

Usage:

```
std::vector<int, SlabAllocator<int, 4096>> v;
```

This ensures:

- Zero fragmentation
- No heap calls
- Optimal cache locality

Constraint: Maximum size must be known.

13.5.4 Reuse-Aware `std::vector` Wrapper

A common reusable pattern is to couple capacity stability with `clear()`:

```
template<class T>
class ReusableVector {
public:
    void reset() noexcept { vec.clear(); }  // keep capacity
    auto& get() { return vec; }
private:
    std::vector<T> vec;
};
```

When used in high-frequency pipelines:

```
thread_local ReusableVector<float> scratch;
auto& buf = scratch.get();
buf.clear();
process(buf);
```

The buffer persists per thread, eliminating allocation churn.

13.5.5 Performance Comparison

Method	Allocation Overhead	Cache Locality	Fragmentatio Risk	Best Use Case
Default `std::allocator`	Medium under reuse; high under churn	Medium	Medium/High	General-purpose workloads
Thread-local reuse (`clear()`)	Very low	High	Low	Hot loops with fixed working set
`std::pmr::monotonic_buffer_resourc`	Zero amortized	High	Region growth only	Batch compute / parsing / simulation frames
Slab/fixed pool allocator	Zero	Maximum	No resizing flexibility	Real-time, embedded, latency-bound tasks

13.5.6 Summary

To optimize `std::vector` reuse patterns:

1. **Retain capacity**, avoid reallocations (`clear()` not `shrink_to_fit()`).
2. Use **thread-local** pools to prevent arena cross-traffic.
3. Prefer **monotonic or region-based allocators** for batch-structured computation phases.
4. Use **slab allocators** in environments requiring predictable latency and hard memory bounds.

Memory allocation is not only a cost center but also a **locality and stability mechanism**.

Performance-critical C++ systems must treat the allocator as a **deliberate architectural choice**, not a passive implementation detail.

Part VI

ELF, LINKER, LOADER, AND BINARY EXECUTION

Chapter 14

ELF Structural Mathematics

14.1 Segment Mapping into Virtual Address Space

When an ELF executable is launched on Linux, the kernel loader constructs the process address space by mapping **program segments** into memory. These segments originate from the **Program Header Table (PHT)**, which describes how the binary is to be realized at runtime. The mapping process establishes the layout of executable code, read-only data, writable data, thread-local regions, the runtime loader, and dynamically linked libraries. Understanding this mapping is fundamental for analyzing performance behavior, memory isolation, address resolution, and binary compatibility across systems.

14.1.1 ELF Segments vs. Sections

At link time, ELF organization is described in terms of **sections** (e.g., `.text`, `.data`, `.rodata`, `.bss`).
At execution time, the kernel ignores sections and uses **segments**, defined in the

Program Header Table:

Structure	Purpose	Used at Compile/Link Time?	Used at Run Time?
Sections	Code & data grouping for the linker	Yes	No
Segments	Memory mapping and permissions	No	Yes

Segments specify ranges of the file that must be mapped into the virtual address space, with associated attributes:

- **Permissions** (R/W/X)
- **Alignment**
- **Offset in file**
- **Virtual address**

The compiler and linker determine how sections are packed into segments, ensuring alignment and page-granular mapping consistency.

14.1.2 Program Header Table (PHT) Structure

The PHT is located near the beginning of the ELF file and contains entries like:

```
Type:  PT_LOAD       # Loadable segment
VirtAddr: 0x400000
FileOffset: 0x000000
FileSize: ...
```

```
MemSize: ...
Flags: R E
Align: 0x200000
```

Typical x86-64 executable layout:

Segment	Purpose	Permissions
Text Segment	Executable code and read-only constants	RX
Data Segment	Initialized writable globals	RW
BSS Segment	Zero-initialized globals	RW (zero-filled at load)
`PT_INTERP` Segment	Dynamic loader path	R
TLS Segment	Per-thread static storage	RW (instantiated per-thread)

14.1.3 Mapping Behavior and Alignment Constraints

Linux uses `mmap` internally to map segments into memory.
Segments are aligned to **page boundaries** (commonly 4 KiB on x86-64) and may be further aligned to **superpages** (e.g., 2 MiB) for TLB efficiency.
Mapping rules:

1. `p_filesz` bytes are loaded from disk.
2. If `p_memsz > p_filesz`, the remaining memory is **zero-initialized** (for `.bss`).
3. Permissions are applied at the page granularity.

This enables efficient memory sharing:

- Code segments are **shared** between processes (copy-on-execute).
- Writable segments are **private**, with copy-on-write behavior.

14.1.4 Address Space Layout and Randomization

Modern Linux systems apply **ASLR (Address Space Layout Randomization)**, relocating:

- Executable base (unless compiled `-no-pie`)
- Shared libraries
- Stack and heap regions

ASLR enhances security by eliminating static address predictability.
Position-independent executables (PIE) are mapped at randomized bases and use `RIP`-relative addressing for functions and data.
Non-PIE executables are mapped at a fixed location (commonly `0x400000`), simplifying static disassembly and debugging at the cost of predictability.

14.1.5 Example: Inspecting Segment Mappings

Using `readelf`:

```
$ readelf -l ./a.out
```

Output excerpt:

```
LOAD   0x000000 0x0000000000400000 0x0000000000400000 R E 0x200000
LOAD   0x001000 0x0000000000600000 0x0000000000600000 RW  0x200000
```

Meaning:

- The first segment maps executable code at virtual address `0x400000`.
- The second segment maps writable data at virtual address `0x600000`.

At runtime, verify with `/proc/<pid>/maps`:

```
$ cat /proc/$(pidof a.out)/maps
```

Expected form:

```
00400000-0040c000 r-xp  ...  a.out   # .text/.rodata segments
00600000-00601000 rw-p  ...  a.out   # .data
00601000-00603000 rw-p  ...          # .bss (zero-filled)
```

14.1.6 Relevance to System-Level C++ Engineering

The way segments are mapped influences:

Design Consideration	**Implication**
Instruction locality	TLB and I-cache efficiency
Data locality	Placement of static/global objects affects cache prefetch paths
Security	ASLR, WX memory enforcement
IPC & Shared Memory	Shared vs private memory pages
Binary compatibility	ABI and runtime loader expectations

C++ runtime behavior, including exception tables, RTTI maps, and vtables, is tightly coupled to the **segments in which they reside**.

14.1.7 Summary

Segment mapping determines how an ELF binary becomes an executing process. It is the **bridge between static layout (link time) and dynamic execution (runtime)**. By understanding the memory permissions, alignment constraints, and the loader's mapping rules, system-level C++ developers can reason about performance behavior, security properties, and ABI conformance with precision.

14.2 Section Grouping, Alignment Models, and Relocation Records

ELF files encode the logical structure of a program through **sections**, which represent arrangement, classification, and linkage of code and data prior to runtime mapping. While segments dictate runtime memory layout, **sections exist primarily for the linker**, symbol resolution, relocation processing, and binary inspection. Understanding how sections are grouped, aligned, and transformed into relocatable program images is fundamental for analyzing C++ compilation, template instantiation artifacts, exception tables, and symbol binding.

14.2.1 Section Grouping and Logical Composition

Sections are grouped to support:

- **Relocation processing**
- **Template and inline-generated code deduplication**
- **Dead code elimination (garbage collection of sections)**
- **Shared object symbol scoping**

ELF uses **comdat groups** to designate sections that may appear multiple times across object files but must be **uniquely selected**. The linker eliminates duplicates based on symbol identity.
Example of group membership:

```
.section .text._ZN1A3fooEv,"axG",@progbits,comdat
.section .data._ZN1A3xEi,"awG",@progbits,comdat
```

The `G` flag declares comdat, ensuring function template instantiations defined in multiple translation units do not produce redundant runtime code.

14.2.2 Alignment Requirements

Alignment ensures predictable address relationships and CPU correctness:

Section	Typical Alignment	Rationale
`.text` (executable code)	16 to 64 bytes	Instruction prefetch / I-cache alignment
`.data` / `.rodata`	8 or natural type alignment	Data load efficiency
`.bss`	Same as `.data`, zero-filled	Page-in on demand
Exception frames (`.eh_frame`)	8 bytes minimum	Unwinding metadata correctness

Linkers may **increase alignment** to satisfy segment boundary requirements.
For performance-tuned builds, code alignment affects:

- Branch prediction efficiency
- Decoding throughput
- Vectorized loop entry alignment

Modern linkers allow explicit alignment control via linker scripts or attributes:

```
__attribute__((aligned(64)))
static int buffer[128];
```

14.2.3 Relocation Records: Type and Resolution Semantics

Relocation records instruct the linker (or dynamic loader) to adjust addresses after symbol placement is finalized. Each relocation entry contains:

- **Offset** into the section where relocation applies
- **Symbol index** referring to a symbol table entry
- **Relocation type** (architecture-specific)
- **Addend** (arithmetic adjustment factor)

For x86-64, common relocation types include:

Relocation	Meaning	Usage
`R_X86_64_PC32`	32-bit PC-relative	Short jumps, local calls
`R_X86_64_PLT32`	Call to symbol via PLT	Dynamic function calls
`R_X86_64_GOTPCREL`	Load address from GOT	PIC data/model access
`R_X86_64_64`	Absolute 64-bit relocation	Non-PIC and kernel code

C++ emits relocation records aggressively in:

- Template instantiation units
- vtable placement
- Typeinfo tables
- Exception unwind metadata

Correct relocation is crucial for:

- Dynamic linking consistency
- Position-independent code correctness
- ABI-stable cross-module symbol references

14.2.4 Interaction with Position-Independent Code (PIC)

Position-independent binaries avoid absolute relocation use by encoding addresses relative to:

- The program counter (`RIP`-relative addressing)
- The Global Offset Table (GOT)
- The Procedure Linkage Table (PLT)

PIC requires that:

```
All global symbol references are resolved through GOT/PLT entries.
```

This enables the loader to relocate the object image without rewriting executable instructions, improving:

- Shared library loading performance
- Memory sharing across processes
- ASLR effectiveness

14.2.5 Example: Inspecting Relocations

Compile a relocatable object:

```
$ g++ -c -fPIC demo.cpp -o demo.o
$ readelf -r demo.o
```

Output example:

```
OFFSET     TYPE              SYMBOL
00000010   R_X86_64_PC32     _Z3foov - 4
```

This indicates that a call to `foo()` is encoded relative to the current instruction pointer and requires placement resolution at link time.

14.2.6 Summary

Concept	Purpose	Relevance to System-Level C++
Section grouping / `comdat`	Eliminates duplicate template instantiations	Reduces binary size; ensures ABI consistency
Section alignment	Controls memory layout for efficient fetch and access	Impacts cache locality and execution throughput
Relocation records	Define how symbol addresses are resolved	Crucial for PIC, shared libraries, and dynamic linking
GOT/PLT indirection	Enables relocation at runtime	Foundation of ASLR and shared object reuse

Section grouping, alignment rules, and relocation semantics define how **compiler output becomes executable machine code**. In advanced C++ system design, these details are not ancillary—they directly influence **performance, binary size stability, ABI integrity, and runtime loader efficiency**.

14.3 Weak, Local, Hidden, Protected, and Global Symbol Rules

Symbol visibility and binding semantics determine how identifiers in an ELF binary participate in linking, relocation, dynamic symbol resolution, and interposition. In C++ programs, the correct handling of symbol visibility is critical for maintaining ABI stability, dynamic library performance, and predictable linkage behavior—particularly given template instantiation, inline emission, RTTI, vtables, and exception type metadata. This section formalizes the properties of **weak**, **local**, **hidden**, **protected**, and **global** symbols under the System V AMD64 ABI.

14.3.1 Symbol Binding Classes

Every symbol in ELF has a **binding** attribute, controlling how it participates in linking:

Binding	Meaning	Resolution Scope	Typical Usage
LOCAL	Not visible outside the object file	Linker-internal only	Static functions, TU-local data
GLOBAL	Publicly visible symbol	Resolved across DSOs	Function exports, shared library API
WEAK	Public but overrideable	Resolved only if non-weak alternative not found	Optional overrides, fallback symbols

Weak binding permits definition override. A weak symbol behaves as:

```
If a strong definition exists → use strong version.
Else → use weak definition.
```

This is used in `libc` for system call wrappers and in C++ libstdc++ fallback routines.

14.3.2 Visibility Attributes and Link-Time Export Control

Visibility controls **dynamic symbol table exposure**, not the existence of the symbol itself:

Visibility	Exposed to DSOs?	Allows Interposition?	Notes
default	Yes	Yes	Standard export; can be overridden
hidden	No	No	Address known at link time; enables direct calls
internal	No	No	Similar to hidden; stronger and link-only
protected	Yes	No	Symbol visible externally but cannot be replaced

In C++:

```
__attribute__((visibility("hidden")))
int internal_state;
```

A hidden symbol allows the compiler to treat references as **local** and avoid GOT/PLT indirection under PIC, improving runtime performance and eliminating interposition ambiguity.
Protected visibility is useful when exporting a symbol without permitting override:

```
__attribute__((visibility("protected")))
void api_function();
```

Call sites use direct addressing, preserving performance while allowing symbol visibility for linking.

14.3.3 Interaction with Position-Independent Code (PIC)

To preserve relocatability, global symbols normally require GOT/PLT lookups:

```
call foo@PLT
```

But if a symbol is marked **hidden**, the compiler replaces the indirect call with `RIP`-relative direct addressing:

```
call foo        # No PLT indirection
```

Meaning:

- `hidden` reduces relocation overhead,
- `hidden` reduces runtime GOT pressure,
- `hidden` improves I-cache performance by eliminating PLT stubs.

Thus, symbol visibility directly affects execution speed and is a **performance tuning parameter**, not just a linkage rule.

14.3.4 Weak Symbols in C++ Object Models

Weak symbols commonly arise in:

1. **Typeinfo (`typeinfo for T`)** and **RTTI**
2. **VTable emission for polymorphic classes**
3. **Template instantiations compiled in multiple TUs**

The Itanium C++ ABI mandates:

- VTables and typeinfo are **weak ODR** symbols.
- The linker selects a single canonical copy during link.
- All references in the program resolve to that canonical copy.

This ensures object identity semantics required for `dynamic_cast` and exception type matching.

14.3.5 Symbol Interposition and Dynamic Linking Behavior

Symbol interposition occurs when a symbol in one DSO overrides another symbol of the same name in a dependent DSO. This only occurs when:

- Symbols are **global**,
- Visibility is **default**,
- The object is dynamically linked.

Interposition is resolved by the dynamic loader based on dependency order (`DT_NEEDED` chain).
Protected and hidden symbols explicitly **disable interposition**, making runtime behavior deterministic.
Example eliminating accidental override:

```
__attribute__((visibility("hidden")))
static int cache_state;
```

14.3.6 Summary

Attribute Type	Governs	Key Behavior	Performance Impact
Binding (`LOCAL`, `GLOBAL`, `WEAK`)	Link selection rules	Weak allows fallback or duplication control	None directly
Visibility (`default`, `hidden`, `protected`)	Dynamic symbol exposure and interposition	Hidden/protected eliminate PLT/GOT indirections	Significant in tight loops and PIC
Weak ODR C++ Symbols	VTables, RTTI, templates	Ensures single canonical definition	Required for ABI correctness

Symbol binding and visibility rules are core to how **C++ semantics are implemented at the binary level**.
They shape:

- ABI stability across shared libraries,
- Dynamic linking behavior,
- PIC performance,
- Identity semantics of polymorphic types and RTTI.

In high-performance C++ system design, symbol visibility should be **consciously designed**, not left implicit.

14.4 DWARF Integration and Line Table Encoding

The DWARF debugging format provides a standardized representation of source-level program structure, type information, control flow, and variable locations that can be mapped back to machine code addresses. In ELF binaries produced by GCC on x86-64 Linux, DWARF metadata resides in dedicated **non-loadable sections**, separate from executable code and data. These sections allow symbolic debuggers, profilers, and sampling tools to reconstruct program semantics without modifying or instrumenting executable code.

DWARF integration is fundamental in areas such as exception unwinding, profile-guided optimization feedback, symbolic stack tracing in runtime diagnostics, and performance tooling workflows.

14.4.1 DWARF Section Structure

Debugging information is primarily stored in the following ELF sections:

Section	Purpose
`.debug_info`	Describes the program structure and type information as DIEs (Debugging Information Entries)
`.debug_abbrev`	Encodes abbreviation tables for compressed DIE representations
`.debug_str`	String table referenced by DIEs
`.debug_line`	**Line number to PC address mapping** tables
`.debug_ranges` / `.debug_rnglists`	Variable and scope address ranges

Section	Purpose
`.debug_loc` / `.debug_loclists`	Variable location expressions
`.eh_frame`	Runtime unwinding metadata (used by exception handling and stack tracing)

DWARF data is read only by debuggers and profiling tools, and **never mapped into executable memory**, preserving security and cache efficiency.

14.4.2 Line Table Encoding Principles

The `.debug_line` section maps machine code addresses to source file line numbers. It encodes:

- **Compilation directory**
- **Source file index table**
- **Instruction address deltas**
- **Line number deltas**
- **Statement boundaries and inlined location entries**

Rather than storing one entry per instruction, DWARF encodes **state transitions** using a compact bytecode interpreted by the debugger. This bytecode updates:

```
(line, column, file, PC address)
```

in a state machine.
This encoding allows efficient compression while retaining precise stepping capability.

14.4.3 Address-to-Line State Machine Encoding

The line table is a sequence of opcodes such as:

Opcode Class	Meaning
Standard Opcode	Adjust address or line, set flags (e.g., `DW_LNS_advance_pc`)
Special Opcode	Combined PC and line delta encoded in one byte
Extended Opcode	Insert markers such as end-of-sequence or define-file directives

Example (conceptual):

```
DW_LNS_set_file     file #3
DW_LNS_advance_pc   +24
DW_LNS_advance_line +2
DW_LNS_copy
```

Debugger interpretation reconstructs a mapping such as:

```
0x400610 → main.cpp : 14
0x400624 → main.cpp : 16
```

Efficient stepping and precise breakpoints rely on this mapping.

14.4.4 Debug Information Entries (DIEs)

`.debug_info` contains a tree of DIEs representing:

- Namespace contexts
- Class definitions, methods, member variables
- Template instantiations (with mangled/linkage names)

- Parameter and local variable scope metadata

Each DIE includes:

- A **tag** (e.g., `DW_TAG_class_type`, `DW_TAG_subprogram`)
- A set of **attributes** (e.g., name, type, address range, linkage name)
- Optional children representing hierarchical structure

C++ template instantiation and overload resolution significantly increase DIE graph complexity, requiring abbreviation tables to avoid redundancy.

14.4.5 Debug vs. Unwind Semantics

The `.eh_frame` section is used at **runtime** by libgcc and libunwind to implement:

- C++ exception propagation
- Stack unwinding for backtraces

Whereas `.debug_frame` (if present) contains equivalent information for **debuggers only** and may use richer encodings not required by the runtime.
Key distinction:

Section	Consumer	Purpose
`.eh_frame`	Runtime	Deterministic unwind during exceptions
`.debug_line`, `.debug_info`	Debugger / Profiler	Source mapping and symbolic analysis

This separation ensures runtime unwind remains minimal, compact, and safe under ASLR.

14.4.6 Practical Inspection

To inspect DWARF line tables:

```
$ objdump --dwarf=decodedline ./a.out
```

To inspect type DIEs:

```
$ readelf --debug-dump=info ./a.out
```

For profiling correlation (e.g., perf + DWARF):

```
$ perf record ./a.out
$ perf report --stdio
```

Line accuracy depends on compiler optimization:

- `-O0` → exact mapping
- `-O2` / `-O3` → instruction scheduling may reduce 1:1 correspondence
- `-fno-omit-frame-pointer` improves stack trace interpretability

14.4.7 Summary

Component	Role	Relevance
`.debug_line`	Maps PC → Source line	Enables debugging, profiling, and code annotation
`.debug_info` (DIE graph)	Encodes program structure	Required for full C++ symbolic debugging

Component	Role	Relevance
`.eh_frame`	Runtime unwinding tables	Required for exception handling and stack tracing
Line state machine encoding	Compact PC/line mapping	Efficient and scalable across translation units

DWARF provides the **semantic bridge** between optimized machine code and source-level program understanding.
For system-level C++ engineering, DWARF metadata is essential for:

- Performance attribution
- Exception diagnostics
- Low-overhead runtime observability
- ABI stability verification

14.5 Examples: Re-mapping ELF Segments via Custom Linker Script

Control over ELF segment layout allows system-level C++ developers to influence memory mapping, cache behavior, load-time locality, and binary security posture.
The GNU linker (`ld`) supports **linker scripts** which explicitly define how sections are grouped into segments, where segments are placed in the virtual address space, and how alignment relationships are enforced.
This capability is essential when:

- Constructing **position-dependent executables** for embedded systems.
- Aligning `.text` to **large page boundaries** to reduce TLB pressure.
- Co-locating data structures to improve **cache locality** in high-performance code.
- Enforcing **W^X** (write-xor-execute) security constraints strictly.
- Reducing page working-set footprint in memory-constrained environments.

14.5.1 Linker Script Core Structure

A minimal script defines how input sections map to output sections:

```
ENTRY(_start)

SECTIONS {
    /* Map .text at fixed aligned base */
    .text 0x400000 ALIGN(0x200000) : {
        *(.text .text.*)
    }
```

```
    /* Read-only data tightly following code */
    .rodata ALIGN(64) : {
        *(.rodata .rodata.*)
    }

    /* Writable data and zero-fill region */
    .data ALIGN(4096) : {
        *(.data .data.*)
    }

    .bss : {
        *(.bss .bss.* COMMON)
    }
}
```

Key properties:

- **`ENTRY(_start)`** sets process entry point.
- **Alignment** ensures page or cache boundary placement.
- Multiple input sections merge into unified logical output regions.

This output section mapping is subsequently converted into **PT_LOAD segments** by the linker.

14.5.2 Controlling Segment Formation

To explicitly influence runtime mapping, we declare **PHDRS** blocks:

```
PHDRS {
    text PT_LOAD FLAGS(R X);
```

```
    data PT_LOAD FLAGS(R W);
}

SECTIONS {
    .text : {
        *(.text*)
    } : text

    .rodata : {
        *(.rodata*)
    } : text

    .data : {
        *(.data*)
    } : data

    .bss : {
        *(.bss*)
    } : data
}
```

This forces `.text` and `.rodata` into a **single RX segment**, improving I-cache prefetch and reducing TLB coverage.
`.data` and `.bss` reside in a **separate RW segment**, preventing accidental execution permissions.
This separation reflects a strict **W^X discipline**.

14.5.3 Example: Large Page Alignment for Instruction Fetch Efficiency

High-throughput HPC or signal-processing loops benefit from mapping `.text` on 2 MiB hugepages to reduce TLB misses.

Link script modification:

```
.text 0x400000 ALIGN(0x200000) : {
    *(.text .text.*)
}
```

And runtime mapping (system config allowing):

```
$ echo madvise | sudo tee /sys/kernel/mm/transparent_hugepage/enabled
$ madvise(text_region, size, MADV_HUGEPAGE);
```

The result: larger page granularity reduces **ITLB pressure** and improves predictable instruction delivery.

14.5.4 Example: Isolating a Hot Data Region Near Executable Code

A high-frequency data structure (e.g., jump tables, frequently-read state arrays) may be deliberately mapped adjacent to `.text`:

```
.hotdata ALIGN(64) : {
    KEEP(*(.hotdata))
}
```

When placed immediately after `.text`, the processor benefits from **spatial locality in unified I/D L1 cache**, reducing long-latency reloads.
To place such globals:

```
__attribute__((section(".hotdata")))
static uint64_t state_buffer[256];
```

14.5.5 Verifying Segment Mapping

After linking:

```
$ readelf -l app | grep LOAD
```

Example output:

```
LOAD 0x000000 0x00400000 0x00400000 R E 0x200000
LOAD 0x001000 0x00600000 0x00600000 R W 0x200000
```

Validate runtime layout:

```
$ cat /proc/$(pidof app)/maps
```

Confirm `.text` and `.rodata` share an RX region, and `.data + .bss` are separate RW.

14.5.6 Summary

Control Mechanism	Purpose	Resulting Property
Section-to-segment grouping	Defines memory protection regions	Enforces W^X and interposition predictability
Alignment directives	Align code and data on cache / TLB boundaries	Reduces ITLB/DTLB churn; improves locality
Custom placement of hot data	Co-locates computation and state	Reduces memory latency in hot loops

Control Mechanism	Purpose	Resulting Property
PHDRS mapping	Explicit control over load segments	Predictable binary and loader behavior

By re-mapping ELF segments via custom linker scripts, a C++ system architect can **reshape memory semantics to match execution behavior**, achieving improvements in:

- Latency stability
- Instruction and data locality
- Security posture
- Binary reproducibility and ABI reliability

This represents the point where **compiler output becomes micro-architectural strategy**.

Chapter 15

Dynamic Loader Algorithm and GOT/PLT Behavior

15.1 Lazy vs Immediate Binding Resolution State Machine

Dynamic linking in ELF environments allows symbols defined in shared libraries to be bound to references in executable code **at runtime**. The dynamic loader (`ld.so`) coordinates this process using the **Global Offset Table (GOT)** and the **Procedure Linkage Table (PLT)**. Program behavior, startup latency, memory locality, and runtime determinism depend on whether symbol resolution is performed **lazily** (when first called) or **immediately** (at application startup).
Understanding the binding state machine is essential for system-level C++ applications where call-path determinism, latency predictability, and dynamic linking correctness must be guaranteed.

15.1.1 PLT/GOT Indirection Overview

For a call to a function in a shared object, the compiler generates an indirect call via the PLT:

```
call foo@PLT
```

The PLT entry contains code that jumps through GOT entries to determine the actual address of `foo()`. Initially, the GOT entry does not contain the final address; it contains a pointer back to the dynamic loader's resolution handler.
At a high level:

Element	Purpose
PLT	Dispatches function calls that may require dynamic resolution
GOT	Stores resolved runtime addresses of external symbols
Resolver	Performs lookup, relocation, and GOT patching

15.1.2 Lazy Binding State Transition

Lazy binding defers symbol resolution until **first call**, reducing startup overhead and enabling shared libraries to be loaded without immediate symbol patching.
State machine (simplified):

```
Initial Call:
    PLT entry → jump to resolver stub → call into ld.so
    ld.so:
        * Lookup symbol in symbol tables
        * Resolve symbol address
        * Patch GOT entry with resolved address
    Return to call site
```

```
Subsequent Calls:
    PLT entry → jump directly to resolved function (no loader invocation)
```

This results in a **one-time resolution penalty per dynamic symbol**, amortized across program execution.

Advantages:

- Lower startup cost
- Lower memory footprint at launch
- Supports dlopen/dlclose flexibility

Disadvantages:

- First-call latency spike
- Loader invocation during execution
- Increased branch unpredictability during initial execution

15.1.3 Immediate Binding State Transition

Immediate binding resolves all symbol references **before** program execution enters `main()`.

Enabling immediate binding:

```
$ LD_BIND_NOW=1 ./app
```

or linking with:

```
-Wl,-z,now
```

State machine:

```
Program Load:
    ld.so:
        * For each PLT-referenced symbol:
            - Perform lookup
            - Patch GOT entry
    main() executes with all call targets fixed
```

Advantages:

- No runtime resolution latency
- Deterministic call behavior
- Stable performance characteristics in tight loops

Disadvantages:

- Longer startup time
- Increased page faults at startup
- Higher memory working-set at load time

15.1.4 Performance and Determinism Trade-offs

Property	Lazy Binding	Immediate Binding
Program startup	Fast	Slower
First call performance	Unpredictable	Deterministic

Property	Lazy Binding	Immediate Binding
Runtime latency	Possible spikes	Stable
Real-time suitability	Poor	Good
Debuggability	More complex	Straightforward
Security (RCE hardening)	Weaker	Stronger (reduces attack window)

For **real-time processing, low-latency trading, simulation loops, HPC kernels**, immediate binding is preferable to avoid path-length variability.
For **general GUI applications, dynamic plugin systems, or interactive scripting runtimes**, lazy binding reduces load time overhead.

15.1.5 C++ Language-Level Effects

Lazy binding interacts with:

- Virtual dispatch (indirect calls stack with PLT indirection)
- Exception unwinding (handler tables are unaffected but tracing overhead may differ)
- Function pointer identity (indirection affects equality and pointer comparison only during unresolved states)

Critically, **pure virtual call resolution and ABI vtable layout** are independent of dynamic binding; PLT only affects symbol linkage, not dispatch semantics internal to object models.

15.1.6 Summary

Component	Role	Implications
PLT	Dispatches calls requiring runtime binding	Introduces indirection on external calls
GOT	Holds resolved symbol addresses	Patched on first use (lazy) or load (immediate)
Lazy Binding	Defers resolution to first call	Minimizes startup cost; introduces runtime latency
Immediate Binding	Resolves all calls before `main()`	Maximizes determinism; increases startup cost

The lazy vs immediate binding model is a **runtime contract** influencing performance predictability and system behavior.
In advanced C++ system design, selecting binding mode is not a configuration detail—it is a **first-order architectural decision** tied directly to execution environment constraints.

15.2 IFUNC, Symbol Interposition, and Auditing Interfaces

The ELF dynamic linking model allows flexible symbol resolution at runtime. Beyond standard lazy and immediate binding, Linux provides **indirect functions (IFUNC)**, **symbol interposition**, and **audit interfaces** to modify or observe symbol resolution. These features are powerful tools for performance adaptation, ABI compatibility layers, profiling frameworks, and hardened security monitoring, but they must be used selectively due to their impact on determinism, caching, and execution transparency.

15.2.1 IFUNC (Indirection Functions) Resolution Mechanism

An **IFUNC** is a symbol whose address is determined by executing a resolver function at runtime. This allows **CPU feature–adaptive implementations** without requiring multiple shared libraries or runtime dispatch branches inside hot code.
Example declaration:

```
extern "C" void foo_impl_ssse3();
extern "C" void foo_impl_avx2();

static void* foo_resolver() {
    if (__builtin_cpu_supports("avx2"))
        return (void*)foo_impl_avx2;
    else
        return (void*)foo_impl_ssse3;
}

extern "C"
__attribute__((ifunc("foo_resolver")))
void foo();
```

At load time (or if lazy-bound, at first call), the dynamic loader executes `foo_resolver()` and patches the GOT entry with the returned function pointer. Subsequent calls incur **zero dispatch overhead**.

Advantages:

- Branch-free CPU specialization
- Compatible with PIC and shared libraries
- No runtime overhead after resolution

Costs:

- Resolver executes during relocation
- Must avoid heavy computation in resolver
- Increases loader complexity if used excessively

IFUNC is widely used in optimized `glibc`, OpenSSL, BLAS libraries, and HPC math kernels.

15.2.2 Symbol Interposition and Resolution Ordering Rules

Symbol interposition allows a symbol in one object to **override** a symbol of the same name in another object, provided:

- Both are **GLOBAL** binding symbols
- Visibility is **default**
- Resolution occurs through **dynamic linking**

Search order:

```
1. Executable (if dynamic)
2. Libraries in DT_NEEDED order, breadth-first
3. LD_PRELOAD libraries (prepend override)
```

Example interposition:

```
extern "C" void foo() {
    // replacement implementation
}
```

And run:

```
$ LD_PRELOAD=./override.so ./app
```

Interposition rewrites call targets by altering PLT/GOT bindings.

Risks in high-performance C++ systems:

- Loss of inlining and devirtualization opportunities
- Indirect call overhead persists even after binding
- Debugging complexity increases significantly

Modern performance-guided C++ designs often mark internal APIs as:

```
__attribute__((visibility("hidden")))
```

to **disable interposition** where determinism is required.

15.2.3 Protected Visibility and IFUNC Interaction

When applying:

```
__attribute__((visibility("protected")))
void foo();
```

the symbol remains externally visible but **is not interposable**.
Combined with IFUNC, this guarantees:

- CPU-optimized implementation selected at load time
- No PLT indirection
- No interposition override vulnerabilities

This is the recommended model for **high-throughput shared library APIs**.

15.2.4 LD_AUDIT and Dynamic Linking Auditing Interfaces

The loader provides an **auditing API** allowing shared libraries to intercept and log:

- Symbol lookup
- PLT/GOT binds
- Object loading and unloading events

To enable:

```
export LD_AUDIT=./audit.so
```

An audit module implements callbacks such as:

```
extern "C" unsigned int la_version() { return LAV_CURRENT; }

extern "C" void* la_symbind64(
    Elf64_Sym* sym, unsigned int ndx,
    void* refcook, void* symcook,
    const char* symname)
{
    // Inspect or log binding
    return (void*)sym->st_value; // Return target address
}
```

Use cases:

Category	Purpose
Profiling frameworks	Record dynamic call graphs
Security monitoring	Detect unauthorized symbol redirection
ABI transition support	Rewrite symbol references across library versions

Auditing runs **inside the loader**, so it must be minimal, thread-safe, and free of recursion into dynamic lookup.

15.2.5 Performance and Security Considerations

Mechanism	Strengths	Costs	Recommended Use
IFUNC	Zero-overhead CPU feature specialization	Resolver overhead at load; added complexity	Numerical kernels, cryptographic fast paths

Mechanism	Strengths	Costs	Recommended Use
Symbol Interposition	Runtime override flexibility	Prevents inlining; introduces call indirection	Testing layers, debugging shims, compatibility layers
Protected / Hidden Visibility	Eliminates binding variability	Must be designed into the ABI early	Library internal APIs, high-performance compute, real-time systems
`LD_AUDIT`	Full symbol-level observability and tracing	High runtime introspection overhead	Debugging, ABI migration verification, runtime conformance checking

15.2.6 Summary

The dynamic linking model is not limited to binding addresses—it is a programmable, policy-controlled resolution system.

- **IFUNC** selects optimized implementations based on microarchitectural capabilities.
- **Interposition** enables override-based layering but must be avoided in latency-critical paths.
- **Protected visibility** guarantees stability and eliminates unnecessary indirection.
- **Audit interfaces** provide controlled introspection into runtime symbol behavior.

In advanced C++ system design, these mechanisms enable strategic control over the **mapping between binary structure and execution performance**, forming a bridge between compilation artifacts, the loader's resolution engine, and microarchitectural optimization behavior.

15.3 RELRO, BIND_NOW, PIE Hardening Behavior

Modern ELF loaders support several binary hardening features that strengthen the runtime memory model against corruption, dynamic call target manipulation, and code-reuse attacks. The principal mechanisms relevant to system-level C++ binaries are:

- **RELRO (Relocation Read-Only)** — restricts mutation of relocation-resolved tables.
- **BIND_NOW** — enforces early binding to eliminate runtime PLT resolution trampolines.
- **PIE (Position Independent Executable)** — enables full Address Space Layout Randomization (ASLR) for the main executable.

Each feature influences the structure and stability of dynamic linking, loader control flow, and GOT/PLT indirection patterns, with effects on both **security** and **execution determinism**.

15.3.1 RELRO: Read-Only Relocation Protection

During dynamic symbol resolution, the dynamic loader writes resolved addresses into:

- The **GOT** (Global Offset Table),
- The **Dynamic Relocation sections**, and
- Various runtime linker state structures.

By default, these memory regions remain writable throughout execution, enabling attack vectors such as:

- GOT overwrite → redirect function call instructions,
- PLT entry hijacking → arbitrary code redirection.

RELRO mitigates this by transitioning relocation-resolved sections to **read-only** after their initialization phase.
RELRO modes:

Mode	Behavior	Protection Coverage	Cost
partial RELRO	`.got.plt` remains writable	Linker metadata only	No runtime overhead
full RELRO	Entire GOT becomes read-only after relocation	Complete GOT protection	Requires immediate binding

Full RELRO requires `BIND_NOW`, since lazy-binding depends on the ability to modify GOT at call time.

15.3.2 BIND_NOW: Immediate Symbol Resolution Enforcement

The `BIND_NOW` option forces the loader to resolve **all PLT entries before `main()` executes**, eliminating `dl-runtime` lazy binding stubs.
Compile-time or link-time selection:

```
-Wl,-z,now
```

or runtime override:

```
LD_BIND_NOW=1
```

Effects:

- Ensures **GOT is stable and safe to make read-only** (required for Full RELRO).
- Eliminates per-symbol first-call latency.
- Produces deterministic call-site behavior across all control paths.

Trade-offs:

Benefit	Cost
Deterministic call performance	Higher startup time
Required for security-hardening	More relocations during load
No runtime resolver entry points	Larger working set at program start

This model is appropriate for **servers, daemons, financial engines, HPC pipelines**, not latency-sensitive short-lived CLI tools.

15.3.3 PIE: Position Independent Executable and ASLR Enforcement

PIE compiles the main executable as position-independent code, enabling it to be mapped at randomized addresses in memory. This disrupts absolute code and data offsets that would otherwise be stable across runs.
Compile:

```
-fPIE -pie
```

Effects:

- All code uses **RIP-relative addressing**, same as shared libraries.
- The **base address of the executable is randomized** at load time.
- Increases resistance to return-oriented programming (ROP) and jump-oriented programming (JOP) attacks.

Memory model impact:

Region	Without PIE	With PIE
`.text`	Loaded at fixed virtual address	Randomized per execution
`.data` / `.bss`	Fixed offsets	Offset relative to randomized program base
GOT / PLT	Fixed tables	Still present, but base address varies

PIE pairs naturally with **Full RELRO + BIND_NOW**, creating a memory environment where code and call indirection targets are both **unpredictable** and **immutable**.

15.3.4 Combined Hardening Model

The strongest security profile uses:

```
-fPIE -pie -Wl,-z,relro -Wl,-z,now
```

Resulting properties:

Property	Security Effect	Performance Impact
PIE	ASLR enabled for executable	Minor for well-optimized PIC
Full RELRO	GOT completely read-only	Requires immediate binding
`BIND_NOW`	No lazy binding or GOT patching	Slightly slower startup; stable runtime
No writable code pointers	Eliminates PLT/GOT overwrite vectors	None at steady state

From the system-engineering perspective, this configuration trades **startup time** for **maximum runtime stability and predictability**.

15.3.5 Summary

Mechanism	Purpose	Strength	When to Use
RELRO	Make relocation tables read-only	High (Full), Medium (Partial)	Always for production binaries
BIND_NOW	Resolve PLT entries at load time	High determinism; required for Full RELRO	Daemons, long-running, or real-time systems
PIE	Enable ASLR for main executable	High entropy & exploit resistance	All modern Linux deployments

In well-architected C++ systems—particularly those that must be safe, deterministic, and resilient to memory corruption—**compiler and linker hardening flags are not**

optional. They define the structural security boundary within which all higher-level abstractions operate.

15.4 GOT/PLT Entry Address Calculation and Trampoline Jump Flow

The interaction between the Global Offset Table (GOT) and the Procedure Linkage Table (PLT) defines how dynamically linked functions are invoked on x86-64 Linux. The compiler emits indirect call instructions referencing PLT entries, while the PLT entries themselves perform indirect jumps using addresses stored in the GOT. These structures enable dynamic symbol resolution, deferred binding, and symbol interposition without modifying instruction encodings in-place. This section formalizes the execution flow for PLT-dispatched calls and the mechanics of GOT patching under lazy and immediate binding modes.

15.4.1 Structural Relationship Between PLT and GOT

Each dynamically linked function call site compiled as:

```
call foo@PLT
```

references a **PLT stub**. That stub:

1. Loads a function address from a **GOT entry**.
2. Indirectly jumps to that function.

GCC and the linker generate:

```
.text:
foo@PLT:
    jmp *GOT[foo]              # indirect jump via GOT
    pushq <relocation-index>   # resolver argument
    jmp PLT[0]                 # call dynamic resolver
```

Thus:

- PLT performs **dispatch**.
- GOT holds **resolved function addresses**.
- The dynamic resolver patches the GOT pointer to replace the trampoline.

15.4.2 Initial GOT State and Lazy Binding Control Flow

Before symbol resolution:

```
GOT[foo] → PLT resolver stub (dl-runtime)
```

Execution flow on first call:

```
call foo@PLT
 → PLT[foo]:
       jmp *GOT[foo]                ; jumps to resolver
 → Resolver trampoline:
       push relocation index
       jmp PLT[0]
 → PLT[0]:
       jmp *GOT[linker resolver]    ; enters ld.so
 → ld.so:
       Resolve symbol
       Patch GOT[foo] = real foo address
 ← Return to caller
```

Second and subsequent calls:

```
call foo@PLT
 → PLT[foo]:
       jmp *GOT[foo]  ; now points directly to foo
```

The trampoline path executes only once.

15.4.3 Immediate Binding Behavior

When `BIND_NOW` is enabled, symbol resolution is performed during program startup:

- The loader resolves all PLT/GOT pairs before `main()`.
- GOT entries contain **final function addresses** prior to execution.
- The resolver path is **never executed** at runtime.

Execution becomes:

```
call foo@PLT
 → PLT[foo]:
       jmp *GOT[foo]  ; already patched, no resolution cost
```

This eliminates first-call latency and supports **Full RELRO** (GOT read-only).

15.4.4 Code Generation Constraints: RIP-Relative GOT Access

In x86-64 System V ABI, the GOT is addressed via **RIP-relative addressing**, ensuring PIC/PID compatibility:

```
jmp *foo@GOTPCREL(%rip)
```

The compiler expresses addresses symbolically; the linker resolves them to GOT-relative displacements:

```
GOT[foo] = &foo (once resolved)
```

This allows:

- PIE executables to be relocated freely.
- Shared libraries to operate with a uniform addressing model.
- No absolute relocation rewriting required.

15.4.5 PLT[0] and the Dynamic Resolver Interface

PLT entry zero (`PLT[0]`) is a special dispatcher used by all unresolved PLT entries:

```
PLT[0]:
    push GOT[1]     ; pointer to dynamic linker's link_map
    jmp GOT[2]      ; jump to resolver entry
```

`GOT[1]` and `GOT[2]` are populated at load time:

- `GOT[1]` → `link_map` describing loaded objects.
- `GOT[2]` → `ld.so`'s resolution entry function (`_dl_runtime_resolve`).

This defines the **ABI contract** between generated PLT code and the system dynamic loader implementation.

15.4.6 Summary of Trampoline Jump Flow

Phase	GOT Entry Value	PLT Behavior	Result
Before resolution	Pointer to resolver stub	PLT jumps to resolver	Loader resolves target
During resolution	GOT patched	PLT path is stable but still indirect	GOT now holds final function address
After resolution	Pointer to final function	PLT jump goes directly to function	Zero overhead dispatch

This incremental state transition ensures:

- **Correctness** with dynamic symbol lookup.
- **PIC/PIE compatibility** through RIP-relative addressing.
- **Optional determinism** via `BIND_NOW`.
- **Hardening** with RELRO to seal GOT after resolution.

15.4.7 Summary

The GOT/PLT dispatch mechanism is a carefully engineered dynamic linkage model where:

- The **PLT mediates call dispatch** through symbolic indirection.
- The **GOT stores resolved addresses**, patched by the dynamic loader.
- **Lazy binding defers cost** but introduces first-use latency and mutability.
- **Immediate binding enforces determinism** and enables hardening (Full RELRO).
- RIP-relative addressing ensures **binary relocatability** under PIE/ASLR.

In high-performance C++ systems, understanding this flow is essential for reasoning about call latency, security exposure, and code layout behavior at runtime.

15.5 Examples: Breakpointing PLT Resolver Inside `ld.so`

Understanding dynamic symbol resolution requires observing the control transfer from the Procedure Linkage Table (PLT) to the dynamic loader runtime (`ld.so`). Instrumenting this path provides direct visibility into lazy-binding behavior, GOT patching, and symbol lookup mechanics. This section demonstrates how to set breakpoints at the PLT resolver entrypoint, inspect GOT updates, and trace symbol resolution flow in a running dynamically linked C++ executable.

15.5.1 Identifying the Resolver Entry Symbol

The dynamic loader exports internal entry points responsible for PLT resolution. On x86-64 glibc systems, the resolver is typically:

```
_dl_runtime_resolve
```

or, depending on glibc version:

```
_dl_runtime_resolve_xsave
```

We inspect available symbols using:

```
$ nm -D /lib64/ld-linux-x86-64.so.2 | grep resolve
```

This yields the resolver's load address offset relative to the interpreter segment.

15.5.2 Launching the Example Target

Consider a program that calls a shared library function:

```
// foo.cpp
#include <iostream>
extern void bar();
int main() {
    bar();
    return 0;
}
```

Compile and link dynamically:

```
$ g++ -O2 -fPIE -pie foo.cpp -L. -lbar -o foo
```

Do **not** enable `-Wl,-z,now`; we want lazy binding to trigger the resolver.

15.5.3 Attaching a Breakpoint in GDB

Launch under gdb with loader symbols loaded:

```
$ gdb foo
(gdb) set stop-on-solib-events 1
(gdb) run
```

Once `ld.so` loads:

```
(gdb) b _dl_runtime_resolve
Breakpoint 1 at 0x7ffff7ddc850
```

Continue execution:

```
(gdb) continue
```

Execution halts when the first unresolved PLT call occurs.

15.5.4 Inspecting Resolver Arguments

Upon hitting the breakpoint:

```
(gdb) info registers rdi rsi
```

Resolver calling convention (System V AMD64):

Register	Meaning
`rdi`	`link_map*` describing the loaded shared-object dependency graph
`rsi`	Relocation index into `.rela.plt`

To inspect the resolved relocation entry:

```
(gdb) x/4gx *(Elf64_Rela *)(link_map->l_info[DT_JMPREL]->d_un.d_ptr + rsi *
↪  sizeof(Elf64_Rela))
```

This yields:

- Symbol table index,
- Offset of GOT entry,
- Relocation type (`R_X86_64_JUMP_SLOT`).

15.5.5 Watching GOT Patching

Before resolution:

```
(gdb) x/gx foo@GOT
0x0000000000404018:  0x00007ffff7ddc850   # resolver entry
```

Step through resolution:

```
(gdb) stepi   # step a few instructions
```

After return, inspect again:

```
(gdb) x/gx foo@GOT
0x0000000000404018:  0x00007ffff7bc1230   # actual bar() address
```

The GOT entry now contains the final function pointer. Subsequent PLT calls are direct jumps to this address.

15.5.6 Verifying PLT → GOT → Function Flow

Disassemble the PLT entry:

```
(gdb) disassemble foo@plt
```

Expect:

```
foo@plt:
    jmpq   *0x404018(%rip)    # jump via GOT[foo]
    pushq  $index
    jmpq   plt[0]
```

Now confirm direct dispatch:

```
(gdb) break bar
(gdb) continue
```

Execution now enters `bar()` immediately, without returning to the resolver pathway.

15.5.7 Interpretation

The observed behavior illustrates the state transitions described previously:

Phase	GOT Entry	PLT Path Execution	Loader Involvement
Before first call	Points to resolver stub	PLT jumps to resolver	Resolution occurs
After first call	Contains real symbol address	PLT jumps directly to function	Loader no longer involved

The non-recurring rewrite of the GOT entry enforces latency amortization across calls. Under `BIND_NOW`, this entire dynamic trampoline execution path is eliminated before `main()`.

15.5.8 Summary

This procedure demonstrates:

- The exact branch path taken during lazy binding.
- How `ld.so` uses relocation index metadata to resolve function targets.
- Real-time GOT patching behavior and its effects on subsequent call performance.
- The difference between first-call resolution cost and steady-state dispatch.

Understanding and being able to **observe** this behaviour is a prerequisite for reasoning about:

- Security hardening (RELRO, BIND_NOW, PIE),

- Control-flow integrity models,
- Performance sensitivity in code with frequent external symbol calls.

Part VII

DEBUGGING, PROFILING, VERIFICATION, AND PERFORMANCE ENGINEERING

Chapter 16

GDB for C++ ABI State Analysis

16.1 Unwinding Optimized Frames Lacking Symbol Boundaries

When debugging optimized C++ binaries, particularly those compiled with `-O2` or `-O3`, frame unwinding is frequently obstructed by the absence of conventional frame boundaries. This occurs because the compiler performs aggressive inlining, register allocation, tail-call elimination, and frame-pointer omission (`-fomit-frame-pointer` default on x86-64 for optimized builds). As a result, the call stack no longer forms a strictly nested chain of frame pointers; instead, execution contexts are implicitly reconstructed from DWARF Call Frame Information (CFI) and register spill heuristics. This section formalizes the mechanics and failure cases of unwinding optimized frames, with emphasis on ABI constraints, compiler lowering decisions, and debugger-side recovery algorithms.

16.1.1 FP and CFA: Distinct Logical Models

Unwinding relies on the **Canonical Frame Address (CFA)**, defined relative to a stable stack reference. In unoptimized code, the CFA is commonly derived from the frame pointer register (`rbp`), forming a simple parent-link chain. Under optimization, the compiler is free to:

- Reuse `rbp` for general-purpose allocation,
- Eliminate the frame pointer entirely,
- Depend on the stack pointer (`rsp`) as the sole frame reference.

Therefore:

```
CFA = rsp + offset    (optimized)
vs.
CFA = rbp + offset    (non-optimized)
```

GDB must track the CFA from DWARF `.eh_frame` or `.debug_frame` tables rather than assuming `rbp` chain validity.

16.1.2 Inlining and Loss of Explicit Call-Site Boundaries

Inlining removes call/return boundaries in the generated machine code. The debugger reconstructs inlined call contexts symbolically through DWARF inline info tables, which describe:

- Original call site location,
- Containing function lexical scope,
- Variable location expressions relative to CFA or registers.

However, inlined contexts **do not imply recoverable runtime frames**. They exist only as symbolic overlays on top of a single physical frame. Stack traces in optimized binaries therefore mix:

- **Physical frames** (real stack)
- **Inline-expanded logical frames** (DWARF-only metadata)

This distinction is essential when interpreting backtraces involving template instantiations and concept-substituted lambdas.

16.1.3 Tail-Call Elimination and Frame Collapsing

Tail-call optimization transforms:

```
caller → call callee → return
```

into:

```
caller → jump callee
```

which removes the caller's runtime frame entirely. This is legal under the System V C++ ABI when:

- Argument conventions match or can be adapted at zero cost,
- No local destructors must run after the call,
- No exception landing pads depend on the outgoing frame.

For a debugger, this means the call chain is **semantically present**, but **physically absent**. GDB reconstructs this relationship using:

- DWARF `.debug_info` call-site metadata (if present),
- Heuristic inspection of PC-to-function-symbol mapping,
- Inline call-tree annotation as fallback.

Tail-call elimination thereby produces valid but non-intuitive backtraces that omit intermediate logical calls.

16.1.4 Register-Allocated Variables and Unwind State Instability

In optimized code, many variables do not reside in memory:

- Scalar locals may live entirely in registers,
- Temporaries may be materialized only across partial basic blocks,
- Liveness intervals may overlap and shift based on branch prediction.

DWARF location expressions annotate register-based variable lifetime:

```
DW_OP_regN                # lives fully in register
DW_OP_bregN + offset      # relative to base register
DW_OP_fbreg + offset      # relative to CFA
```

If the variable's register is repurposed or clobbered at the sampling point, GDB reports it as **optimized out**, even though it was logically present at compile time. This is not a debugger limitation; it reflects the fundamental transformation rules of SSA and register allocation.

16.1.5 Recovery Strategies in GDB

To improve stack reconstruction accuracy:

- Force frame pointers:

```
$ g++ -O2 -fno-omit-frame-pointer
```

- Preserve variable location clarity:

```
$ g++ -g3 -fvar-tracking-assignments
```

- Enable precise unwind metadata:

```
$ g++ -fasynchronous-unwind-tables -fexceptions
```

For minimal perturbation debugging builds, it is advisable to use:

```
-O2 -g -fno-omit-frame-pointer
```

This retains most optimization benefits while allowing predictable unwinding.

16.1.6 Summary

Optimization Feature	**Effect on Unwind**	**Debugger Recovery Mechanism**
Frame pointer omission	Removes explicit frame chain	CFA via DWARF CFI
Inlining	Eliminates physical call frames	DWARF inline call metadata

Optimization Feature	**Effect on Unwind**	**Debugger Recovery Mechanism**
Tail-call elimination	Removes intermediate frames	Symbol + call-origin analysis
Register allocation	Eliminates stable variable addresses	Variable location expressions

Understanding that optimized stack traces reflect **transformed execution structure**, not **source-level control flow**, is essential when debugging modern C++ binaries. Correct interpretation requires familiarity with both ABI-level calling conventions and compiler lowering strategies, as described in preceding chapters.

16.2 On-the-fly Reconstruction of Object Layout

During debugging of optimized C++ binaries, the debugger frequently encounters object instances whose in-memory representation no longer directly reflects the abstract layout described in the source type. Optimization passes (inlining, scalar replacement of aggregates, dead-store elimination, register promotion, and partial lifetime shortening) often transform an object into a **distributed representation** consisting of:

- Registers holding active fields,
- Spill slots allocated on the stack,
- Temporaries fused into SSA-defined value ranges,
- Or fields wholly eliminated due to proven non-use.

GDB must therefore *reconstruct* the logical object layout at runtime using DWARF type descriptions, location lists, and liveness intervals, rather than assuming a contiguous, stable memory block. This section formalizes how such reconstruction is performed and outlines the conditions under which it fails.

16.2.1 Object Model Stability vs. Optimization-Induced Fragmentation

In the canonical C++ object model (Itanium ABI):

- `struct` and `class` types imply a fixed in-memory field layout,
- Virtual base adjustments are encoded through vtable-relative offsets,
- Derived-to-base pointer conversions are purely offset arithmetic.

Optimization does not change the *abstract layout*, but may aggressively change its *physical realization*.
Example transformations under full optimization:

Field Behavior	Lowering Target	Condition
Scalar field accessed frequently	Promoted to register	Field does not escape and its address is never taken
Field written but never read	Eliminated entirely	Proven dead store
Aggregate copied only once	Scalarized into independent SSA values	Proven independent per-use
Small object passed by value	Materialized in registers	ABI calling convention allows it

Thus, debugging requires **symbolic reconstruction** from metadata.

16.2.2 DWARF Location Lists for Field-Level Resolution

Each field in a class has an associated DWARF location expression describing how to obtain its value at a given program counter (`pc`). Example location descriptors:

- `DW_OP_regN` — field lives entirely in register `N`.
- `DW_OP_fbreg + offset` — field resides at stack offset relative to CFA.
- `DW_OP_bregN + offset` — field resides at memory referenced by register.
- `DW_OP_piece` — field is split into multiple partially overlapping storage fragments.

To reconstruct:

1. Identify the object's static type from symbol information.
2. Compute the CFA from `.eh_frame` unwind tables.
3. Resolve each field's active location expression at the current instruction address.
4. Materialize field values by extracting bytes from registers and memory segments.

Example (conceptual):

```
struct S { int a; double b; };

# a → DW_OP_reg5
# b → DW_OP_fbreg -16
```

Here, `a` resides in `r8d`, `b` resides on the stack.

16.2.3 Composite Object Reconstruction in GDB

When printing a C++ object:

```
(gdb) print obj
```

GDB performs:

- Type query → extract class layout record.
- Field iteration → evaluate each DWARF location expression.
- On failure → annotate field as `<optimized out>`.

If parts of the object are in registers and others in memory, GDB synthesizes a virtual unified view by composing temporary snapshots.
This process is **non-invasive**: no instrumentation, no memory copying back into struct form.

16.2.4 Failure Modes and Non-Recoverability Conditions

Reconstruction fails when:

1. The compiler emits no location record for a field (e.g., field eliminated).
2. The field exists, but its lifetime has ended (liveness interval expired).
3. The function was compiled without DWARF (`-g0` or stripped).
4. The unwind tables cannot identify
   a stable CFA (rare under `-fasynchronous-unwind-tables` but common in JIT or manually patched code).

In such cases, GDB reports:

```
<optimized out>
```

This is **not** incorrect—it reflects the fact that the object does not exist in memory in a materializable form at that point.

16.2.5 Debug Builds for Reliable Object Reconstruction

For high-fidelity debugging of complex object graphs:
Recommended flags:

```
-O2 -g3 -fno-omit-frame-pointer -fvar-tracking -fvar-tracking-assignments
```

These preserve:

- Sufficient unwinding metadata (`-fno-omit-frame-pointer`),
- Precise field-level movement tracking across SSA transformations (`-fvar-tracking-assignments`),

- High debug symbol resolution detail (`-g3`).

This configuration maintains almost all runtime optimization benefits while greatly improving debuggability.

16.2.6 Summary

Component	Role in Reconstruction	Constraint
DWARF type metadata	Defines structural layout	Independent of optimization
Location lists	Map fields to registers/stack segments	Valid only within active liveness ranges
CFI + CFA	Recover base frame location	Requires unwind table integrity
Field liveness	Determines reconstructability	May be zero-length under aggressive optimization

On-the-fly object reconstruction is a **metadata-driven reassembly process**, not a memory dump. Correct interpretation requires understanding that optimized C++ objects frequently **do not exist as contiguous physical entities**. The debugger's role is to logically *reconstruct* them from the SSA-derived storage state.

16.3 Reverse Debugging and Record–Replay Execution

Traditional debugging assumes forward-only progression: the debugger advances execution state while the engineer interprets program behavior. However, optimized C++ binaries often exhibit failure modes that occur significantly after the underlying cause (e.g., heap misuse, stale references, moved-from object usage, or incorrect lifetime assumptions). In such cases, *reverse debugging*—stepping execution backward—provides the ability to observe the exact state evolution leading to failure. This section describes the mechanisms that enable reverse execution in GDB, the architectural constraints of record–replay instrumentation, and the cases where deterministic replay becomes limited by optimization artifacts or hardware interactions.

16.3.1 Determinism Requirements and Sources of Non-Reproducibility

Record–replay execution relies on reconstructing program state by replaying the effects of nondeterministic events. Deterministic replay requires that all external influences be captured. Nondeterministic input sources include:

- System calls returning time, data, or randomness,
- Thread scheduling and synchronization interleavings,
- Hardware exceptions (page faults, signals),
- Memory-mapped I/O and device register interactions.

To achieve reproducibility, record–replay systems intercept and log these nondeterministic operations, producing a serialized event log that can be safely replayed

during debugging. The required granularity differs based on execution model:

Source of Nondeterminism	Logging Strategy
System calls	Record return values and observable side effects
Thread scheduling	Serialize scheduler decisions
Signals	Log signal delivery and context
Shared memory interactions	Require deterministic locking or full-memory logging

Highly parallel code increases log volume; single-threaded code typically yields minimal record overhead.

16.3.2 GDB Process Record / Replay Infrastructure

GDB provides a built-in record–replay engine via `record full` mode:

```
(gdb) record full
```

This activates:

- Instruction-level recording of register and memory stores,
- Event log maintenance in an in-memory ring buffer,
- Reverse execution primitives (`reverse-step`, `reverse-next`, `reverse-continue`).

Backward stepping:

```
(gdb) reverse-step
(gdb) reverse-next
(gdb) reverse-continue
```

GDB reconstructs prior execution states by restoring register sets and memory blocks from the recorded log. When buffer capacity is exceeded, the oldest entries are discarded—record length depends on program state complexity and memory mutation rate.
Strengths:

- Works even on heavily optimized binaries.
- Requires no compiler instrumentation.
- Provides precise historical state reconstruction.

Limitations:

- Increases execution overhead (typically 2× to 15×).
- Memory-intensive for write-heavy workloads.
- Does not capture kernel or device state transitions beyond logged syscalls.

16.3.3 rr: Deterministic Record–Replay for Multi-Threaded C++ Systems

For multi-threaded applications, GDB's built-in recorder may be insufficient due to scheduler nondeterminism. The *rr* tool (userspace deterministic replay engine) provides stronger guarantees:
Execution under `rr record`:

```
$ rr record ./app
```

Replay under GDB with reverse stepping support:

```
$ rr replay
(rr) reverse-next
(rr) reverse-continue
```

Key architectural choices in rr:

- Serializes thread scheduling to ensure deterministic execution.
- Logs only sources of nondeterminism (system call results, signal delivery).
- Avoids full memory logging by replaying instructions exactly.

This makes *rr* suitable for debugging:

- Data-race-induced heap corruption,
- Transient lifetime bugs relating to move semantics,
- Incorrect atomic synchronization patterns.

Performance overhead is moderate (~1.2× to 5× typical).

16.3.4 Memory Model Visibility and C++ Object State Recovery

Reverse execution makes it possible to observe:

- Where a moved-from object lost its last valid value,
- The precise point at which a shared pointer reference count reached zero,
- The moment a stale pointer originated due to container reallocation,
- Cross-thread ownership transfer without synchronization.

When combined with DWARF location tracking (Section 16.2), reverse debugging allows reconstruction of objects *as they existed historically*, not merely at failure time. Key advantage:

```
Instead of debugging where the crash *happened*,
reverse debugging lets us debug where the object *became invalid*.
```

This shifts the debugging paradigm from **post-failure analysis** to **causality tracing**.

16.3.5 Constraints Under Full Optimization

Reverse debugging remains valid when code is optimized, but information loss persists:

Compiler Optimization	Reverse Debug Impact
Inlining	Logical frames reconstructed from DWARF metadata
Register promotion	Values retrievable only during active liveness ranges
Dead-store elimination	Some historical object states never existed materially
Tail-call elimination	Frame collapse reduces visibility of call chain origins

Reverse execution **does not recreate eliminated states**; it navigates the actual lowered execution, not the source-level abstraction. Therefore, interpreting results requires fluency in the lowered IR model (Chapters 5–9).

16.3.6 Summary

Technique	Scope	Strength	Constraint
GDB `record full`	Single-threaded or coarse-threaded workloads	Precise historic reconstruction	High memory and performance overhead
`rr`	Multi-threaded deterministic replay	Low logging cost, stable replay	Serializes scheduling; not suitable for real-time workloads
Reverse stepping	Root-cause tracing backwards from failure	Enables lifetime and ownership debugging	Does not recover compiler-eliminated states

Reverse debugging transforms debugging strategies for optimized C++ binaries: rather than diagnosing effects, the engineer can trace **causal events backward**, recovering execution pathways that conventional debugging cannot reveal.

16.4 Python-Driven Structural Introspection Automation

The GDB Python API enables programmatic inspection of C++ execution state, permitting extraction of type layouts, object field mappings, symbol relationships, register states, and unwind metadata directly from the running process. For modern C++ where compiler optimizations fragment object representations across registers, stack regions, and temporary SSA values, automated introspection is often superior to manual debugging. This section describes the architectural model for Python-driven introspection inside GDB, mechanisms for resolving ABI-layer object relationships, and techniques for constructing reproducible structural analyzers for optimized binaries.

16.4.1 The Python/GDB Integration Model

GDB exposes a Python object hierarchy that mirrors debugger entities:
Using these abstractions allows writing structural analyzers that adapt to compiler-generated layout rather than relying on static assumptions.

16.4.2 Extracting C++ Class Layout from Debug Information

C++ types in DWARF contain:

- Field names
- Field byte offsets
- Subobject inheritance structure
- Virtual base adjustment information

GDB Entity	Python Object Type	Purpose
Frame	`gdb.Frame`	Inspect stack frame, CFA, registers, and program counter (PC).
Value	`gdb.Value`	Represents a typed value; supports dereference, casting, and field/member lookup.
Type	`gdb.Type`	Encapsulates C/C++ type metadata obtained from DWARF debug information.
Symbol Table	`gdb.Symbol` / `gdb.Block`	Resolves symbol names to associated types, memory addresses, and scope locations.
Inference Helpers	`gdb.selected_frame()`, `gdb.lookup_type()`	Dynamic lookup and contextual evaluation within the active debugging session.

Python example:

```
import gdb

def describe(type_name):
    t = gdb.lookup_type(type_name)
    print(f"Type: {t}")
    for f in t.fields():
        print(f"  {f.name}: offset={f.bitpos // 8} bytes, type={f.type}")

describe("std::string")
```

This outputs the *logical* ABI layout even when the object itself is partially optimized out.

16.4.3 Resolving Runtime Object Instances

Given an instance `obj` in the current frame:

```
obj = gdb.parse_and_eval("obj")
```

To iterate fields safely:

```
for f in obj.type.fields():
    try:
        value = obj[f.name]
        print(f"{f.name} = {value}")
    except gdb.error:
        print(f"{f.name} <optimized-out>")
```

This logic matches the symbolic reconstruction model outlined in Section 16.2.

16.4.4 Walking VTables and Virtual Hierarchies

The Itanium ABI prescribes:

- First pointer-sized slot → address of vtable data
- Vtable layout → array of function pointers and RTTI reference

Python inspection:

```
def vptr(obj):
    return int(obj.address.reinterpret_cast(gdb.lookup_type("void").pointer()))

def rtti(obj):
```

```
    return gdb.parse_and_eval(f"*((void**){vptr(obj)} - 1)")

print("vptr:", hex(vptr(obj)))
print("RTTI:", rtti(obj))
```

This resolves:

- Dynamic type identity,
- Virtual dispatch targets,
- Base class orientation.

Such introspection is critical when debug frames are ambiguous due to inlining and devirtualization.

16.4.5 Automating Structural Checks Across Call Frames

Python-driven analysis can traverse multiple frames and evaluate object consistency:

```
frame = gdb.newest_frame()
while frame:
    try:
        locals = frame.block()
        # Example: Detect std::vector resizing events
        for sym in locals:
            if sym.is_argument or sym.is_variable:
                val = frame.read_var(sym.name)
                if str(val.type).startswith("std::vector"):
                    # Inspect size/capacity relationship
                    sz = int(val["_M_impl"]["_M_finish"] - val["_M_impl"]["_M_start"])
                    cap = int(val["_M_impl"]["_M_end_of_storage"] -
                    ↪  val["_M_impl"]["_M_start"])
```

```
                if sz > cap:
                    print(f"[Warning] Vector overflow detected in frame
                    ↪  {frame.name()}")
    except:
        pass
    frame = frame.older()
```

This model scales to automated detection of:

- Dangling references,
- Containers invalidated by reallocation,
- Memory ownership rule violations,
- Incorrect destructor sequencing traces.

16.4.6 Application: Stable Forensic Snapshots Under Reverse Debugging

When combined with reverse execution (Section 16.3), Python introspection scripts can *record semantic object states* over time, not just raw memory content.
Instead of manually stepping backward to locate a corruption point, one can:

- Monitor container invariants,
- Break when invariants fail,
- Dump reconstructable object fragments,
- Continue reversing to the causal write.

This transforms debugging from interactive exploration to *post-hoc structural verification.*

16.4.7 Summary

Feature	Purpose	Benefit
DWARF-driven field iteration	Recover object state regardless of physical layout	Works under compiler optimization.
VTable / RTTI graph inspection	Reveal true dynamic type and dispatch resolution path	Critical for debugging polymorphic runtime behavior.
Frame-automated invariants	Detect semantic corruption instead of only crash symptoms	Enables earlier detection of hidden state errors.
Reverse + Python integration	Supports structural time-travel debugging	Makes isolating the original cause of failure feasible.

Python-driven introspection elevates debugging from memory inspection to **semantic program analysis**, aligning the debugger with the internal C++ object model and ABI behavior discussed throughout this book.

16.5 Examples: Pretty-printing C++ Polymorphic Hierarchies Automatically

Polymorphic class hierarchies are central to the C++ object model, yet under optimization the debugger must infer dynamic type identity through vtable entries, RTTI descriptors, and ABI-defined pointer adjustments rather than relying on static declarations. To improve readability and automate structural inspection, GDB supports Python-driven pretty-printers that interpret polymorphic object state at runtime and render human-meaningful representations. This section demonstrates how to automatically detect dynamic types, traverse base–derived relationships, extract field values regardless of storage location, and display hierarchical information predictably.

16.5.1 Dynamic Type Resolution via the Itanium ABI

Under the Itanium C++ ABI used on Linux/x86-64, every polymorphic object contains a vptr at offset zero:

```
| 0x00 | vptr → vtable → RTTI descriptor → type_info name
| 0x08 | first non-static data field
| ...  | class-defined members
```

Dynamic type identification proceeds as follows:

1. Read the vptr from the object.
2. Resolve the vtable base address.
3. Dereference the RTTI pointer located at `*(vtable - sizeof(void*))`.
4. Interpret the type_info name stored in `.rodata` sections.

GDB already performs this operation when printing polymorphic values with `display /r`, but Python integration allows this mechanism to be embedded in automated formatting routines.

16.5.2 Python Pretty-Printer Registration

GDB discovers pretty-printers through Python modules registered on startup:

```
import gdb.printing

class PolyPrinter:
    def __init__(self, val):
        self.val = val

    def to_string(self):
        dynamic = self.val.dynamic_type
        return f"<{dynamic.tag}> object"

def build_pretty_printer():
    pp = gdb.printing.RegexpCollectionPrettyPrinter("cpp_poly")
    pp.add_printer("polymorphic", ".*", PolyPrinter)
    return pp

gdb.printing.register_pretty_printer(gdb.current_objfile(),
                                     build_pretty_printer())
```

This simple template intercepts printed values and replaces raw addresses with dynamic type names. Real printers extend this behavior to include field decoding.

16.5.3 Hierarchy Expansion Through Base Class Traversal

To produce a structured view of an object and its base classes:

```
def walk_bases(typ):
    yield typ
    for base in typ.fields():
        if base.is_base_class:
            yield from walk_bases(base.type)

def describe_object(obj):
    dynamic = obj.dynamic_type
    result = []
    for t in walk_bases(dynamic):
        result.append(f"[{t.tag}]")
        for field in t.fields():
            if field.is_base_class:
                continue
            try:
                value = obj.cast(t)[field.name]
                result.append(f"  {field.name} = {value}")
            except gdb.error:
                result.append(f"  {field.name} <optimized-out>")
    return "\n".join(result)
```

This function:

- Identifies the dynamic type,
- Walks all public, protected, and private base classes,
- Prints each data field,
- Gracefully handles optimized-out or register-promoted values.

16.5.4 Applying Pretty-Printers Automatically

Inside `.gdbinit`:

```
python
import cpp_poly
end
```

Usage:

```
(gdb) print obj
[Derived]
  value = 42
  mode = ACTIVE
[Base]
  id = 7
```

Even if:

- `Base::id` is spilled to the stack,
- `Derived::mode` is held in a register,
- The object's static type in the current frame is `Base&`,

the script recovers the **true dynamic instance layout**.

16.5.5 Practical Example: Inspecting `std::unique_ptr` to Base

```
struct Base { virtual void f(); int id; };
struct Derived : Base { int value; };

std::unique_ptr<Base> p = std::make_unique<Derived>();
```

At runtime:

```
(gdb) print *p
[Derived]
  value = 12
[Base]
  id = 3
```

Without printers, GDB yields:

```
$1 = {id = 3}
```

or, under optimization:

```
$1 = <synthetic pointer> <optimized-out>
```

The pretty-printer reintroduces **hierarchical semantic meaning** lost during lowering.

16.5.6 Summary

Component	Role	Benefit
vptr + RTTI	Identify dynamic type	Required when static type is not indicative.
Base class tree walk	Reconstruct inheritance hierarchy	Respects C++ ABI class layout rules.
Field enumeration via `DWARF`	Recover data members under optimization	Works even when partially register-bound.
Python custom pretty-printers	Automate structured and readable debug output	Reduces manual symbolic debugging effort.

Automated polymorphic pretty-printing aligns debugger output with the *semantic* structure defined at the C++ source level, while still reflecting the *physical* storage structure defined by the compiler's optimization and ABI-lowering pipeline.

Chapter 17

Performance Profiling and Pipeline Diagnostics

17.1 perf Event Group Models and Event Attribution

Performance analysis on modern out-of-order superscalar x86-64 processors requires interpreting execution behavior in terms of *event groups* rather than individual raw counters. The perf subsystem provides structured grouping of hardware performance monitoring events to allow consistent attribution of pipeline stalls, retirement bottlenecks, cache miss patterns, and branch prediction degradation. Understanding these group models is essential for diagnosing performance anomalies in optimized C++ binaries where compiler transformations obscure direct correlation to source code.

17.1.1 Hardware Performance Counters and Event Domains

Contemporary x86-64 cores expose several key event domains:

Domain	Meaning	Example Signals
Frontend	Instruction fetch and decode behavior	I-cache misses, ITLB misses, decode bandwidth stalls.
Backend	Execution resource pressure and availability	ALU port pressure, store buffer full, ROB exhaustion.
Memory Subsystem	Cache + DRAM interaction behavior	L1/L2/L3 misses, TLB refills, LLC occupancy metrics.
Branch / Control Flow	Branch prediction correctness and stability	Misprediction penalties, BTB conflict rates.
Retirement	Architecturally completed instructions	IPC (instructions per cycle), µops retired.

Raw counters alone are insufficient; attribution requires correlating stalled cycles to the correct causality domain.

17.1.2 Event Grouping: Coordinated Measurement Guarantees

perf's event grouping model ensures that:

- All events in a group **start and stop simultaneously**.
- Group scheduling maintains hardware counter alignment.
- Ratios derived across events within the same group are **architecturally meaningful**.

Example:

```
$ perf stat -e \
  '{cycles,instructions,branches,branch-misses}' \
  ./app
```

Interpreting these events independently would be misleading. Grouping ensures:

```
IPC = instructions / cycles
Branch Mispredict Rate = branch-misses / branches
```

remain valid under multiplexing constraints.

17.1.3 Stalled Cycle Attribution and Pipeline Accounting

Modern CPUs treat cycles where no macro-op retires as *stalled cycles*, but this stall may be caused by different subsystems. perf distinguishes:

Stall Class	Signal	Root Cause
Frontend stall	`idq_uops_not_delivered`	Instruction supply insufficient.
Backend stall	`backend_bound` or port pressure events	Execution resources congested.
Memory bound stall	`l1d_miss`, `l2_miss`, `mem_load_retired.*`	Load latency on the critical path.
Branch stall	`branch-misses` × misprediction penalty	Control-flow resolution delay.

Attribution is therefore *non-distributive*: total stalls sum of stall classes. Instead, stalls must be analyzed via *hierarchical dominance* rules:

1. If pipeline not fed → **frontend bottleneck**.

2. Else if operands unavailable → **memory bottleneck**.
3. Else if execution ports saturated → **backend bottleneck**.
4. Else → **branch speculation or retirement bottleneck**.

This ordering matches the microarchitectural scheduling path.

17.1.4 Event Group Models for Pipeline Diagnostics

Representative event groups used in profiling optimized C++ workloads:

Instruction Throughput Group

```
perf stat -e '{cycles,instructions,task-clock,cpu-clock}' ./app
```

Interpretation:

- IPC close to core-width (e.g., ~4 on Skylake) → near peak throughput.
- IPC < 1.0 → memory-bound or serialized execution.

Frontend Supply Group

```
perf stat -e '{idq_uops_not_delivered.core,icache.misses,itlb.misses.walk.completed}'
↪  ./app
```

Indicates decode starvation or fetch stalls.

Memory Latency Group

```
perf stat -e
↪  '{mem_load_retired.l3_miss,mem_load_retired.fb_full,cycle_activity.stalls_l3_miss}'
↪  ./app
```

Identifies memory-critical-window delay effects.

Execution Port Utilization Group

```
perf stat -e '{uops_executed.port_0,uops_executed.port_1,...}' ./app
```

Reveals port pressure and execution bottleneck alignment.

17.1.5 Attribution to C++ Source Constructs

Once performance bottleneck class is identified, attribution to C++ constructs follows compilation flow:

Profiling Domain	**C++ Construct Likely Responsible**
Frontend bound	Large template instantiations, heavy inline expansion.
Memory latency bound	Non-contiguous containers, pointer-chasing, cache misses.
Backend port pressure	High arithmetic density, instruction scheduling imbalance.
Branch misprediction	Complex conditionals, virtual dispatch without devirtualization.

Significant: attribution requires correlating **binary-level hot paths** (via `perf report` or `perf annotate`) back to the **lowered IR behavior**, not direct source expressions.

17.1.6 Summary

Concept	**Purpose**	**Interpretation Requirement**
perf event groups	Ensure synchronized, comparable measurements.	Ratios are valid only inside event groups.
Stall attribution model	Distinguish root cause versus surface symptom.	Analyze in the hierarchical pipeline model.

Concept	Purpose	Interpretation Requirement
Source-to-microarchitecture mapping	Relate optimized code to performance behavior.	Requires understanding compiler lowering and ABI.

Correct use of perf demands reasoning in terms of **pipeline utilization**, not raw event deltas. Event grouping, attribution hierarchies, and ABI-aware interpretation together yield actionable performance diagnosis, forming the foundation for subsequent chapters on microarchitectural hotspot reduction.

17.2 Branch Mispredict, ROB Stall, RS Full, Store Buffer Full, etc.

Pipeline stalls in modern superscalar out-of-order x86-64 processors arise from interactions between speculative control-flow execution, dependency chains, execution resource limits, and memory subsystem latency. These stalls are not independent; they reflect distinct points in the pipeline where forward progress becomes impossible. Effective performance diagnostics require distinguishing *where* in the pipeline execution is blocked and *why* the stall is structurally dominant. This section formalizes the roles of key pipeline structures—Branch Prediction Unit (BPU), Reorder Buffer (ROB), Reservation Stations (RS), and Store Buffer (SB)—and provides interpretation guidelines for profiling metrics using `perf`, top-down analysis, and microarchitectural event groups.

17.2.1 Branch Misprediction and Control-Flow Recovery

The branch predictor determines which instruction path to fetch speculatively. A mispredicted branch invalidates speculative work:

- All in-flight µops younger than the branch are squashed.
- Pipeline flush and frontend refetch occur.
- Recovery latency typically spans ~**15–22 cycles** on Skylake-class CPUs.
- The mispredicted path stalls **frontend supply** until new target instructions arrive.

Performance counter interpretation:

Metric	Meaning
`branch-misses`	Number of mispredicted branch instructions.
`branches`	Total executed branch instructions.
`branch-misses / branches`	Misprediction rate.
`cycles / branch-misses`	Amortized penalty per misprediction.

High mispredict rates indicate performance tied to unpredictable control flow such as:

- Virtual function dispatch without devirtualization,
- Data-dependent branching (e.g., string scanning),
- Complex decision trees lacking branch-free rewrite opportunities.

17.2.2 ROB Stall: Reorder Buffer Saturation

The Reorder Buffer holds in-flight µops until retirement in program order. When the ROB is full:

- No new µops can issue.
- Execution stops until one or more µops retire.
- ROB full is typically a *symptom*, not a root cause.

Common causes:

Cause	ROB Pressure Source
Long dependency chain on loads	Memory latency stalls retirement.
Independent µops but poor speculation	Flushes prevent progress in retirement.
Integer-heavy loops with unresolved dependencies	Value dependencies block completion and retirement.

Related perf indicators:

```
cpu/event=0xA2,umask=0x01/   # Resource_Stalls:ROB
topdown:backend_bound
```

17.2.3 RS Full: Reservation Station Congestion

Reservation Stations queue µops waiting for operand availability. RS full indicates:

- Ready µops cannot be issued because execution ports are oversubscribed.
- This represents **backend pressure**, not frontend starvation.

Typical patterns:

Pattern	Example C++ Construct
Integer ALU saturation	Tight scalar arithmetic loops.
FP pipeline saturation	Matrix multiply without vectorization.
Misbalanced ILP	Partially vectorized code with underutilized SIMD lanes.

Key measurement:

```
cpu/event=0xA2,umask=0x08/   # Resource_Stalls:RS
```

Mitigation paths include vectorization, unrolling, or explicit scheduling rewrites.

17.2.4 Store Buffer Full: Memory Store Commitment Stall

The Store Buffer holds pending memory stores before they commit to L1. If full:

- No additional stores can issue.
- Loads may stall waiting for store-to-load forwarding correctness checks.

This is common in:

- Producer–consumer pipelines with high write traffic,
- Struct writes that exceed store bandwidth,
- Poorly aligned or uncoalesced writes.

Performance signal:

```
cpu/event=0xA2,umask=0x04/   # Resource_Stalls:SB
```

Remedies include:

- Reducing write rates (e.g., avoid frequent container reallocation),
- Improving alignment,
- Prefetching and write combining where applicable.

17.2.5 Integrating Stall Attribution: Top-Down Microarchitectural Analysis

Intel's Top-Down Method classifies pipeline slots into four mutually exclusive categories:

Class	Interpretation
Retiring	Useful architecturally visible work.
Bad Speculation	Branch mispredictions and speculative pipeline flushes.
Frontend Bound	Instruction supply or decode bandwidth constraints.
Backend Bound	Execution resources or memory subsystem delays.

Stall origin interpretation hierarchy:

```
If bad speculation high → branch predictor tuning / control flow refactoring.
Else if frontend bound → I-cache behavior / inlining choice / code layout.
Else if backend bound and memory-bound → Data structure layout and access locality.
Else if backend bound and core-bound (RS/ROB/SB) → Arithmetic intensity or port
↪  pressure.
```

This hierarchical reasoning avoids misattribution—for example, ROB fullness is a *symptom* of deeper backend stalls, not a cause.

17.2.6 Summary

Stall Type	**Structural Cause**	**Diagnostic Metric**	**Likely C++ Trigger**
Branch misprediction	Control-flow speculation invalidation.	`branch-misses / branches`	Complex branching, polymorphic execution.
ROB full	In-flight retirement blocked.	`resource_stalls.rob`	Dependency chains, cache miss latency.
RS full	Execution ports oversubscribed.	`resource_stalls.rs`	Scalar bottlenecks and weak ILP.
Store buffer full	Commit bandwidth exhausted.	`resource_stalls.sb`	High write traffic or contested shared data.

Pipeline stall classification is critical before attempting optimization. Correct diagnosis requires mapping back to the compiler's lowering decisions and data layout decisions in the C++ codebase.

17.3 Flame Graph Construction and Cycle Attribution

Flame graphs provide a visual representation of where execution time (or sampled cycles) is spent across the call stack. Unlike raw performance counters, flame graphs express *hierarchical cost attribution*—they show *which* functions consume cycles and *how* they were reached. This is essential in optimized C++ systems where aggressive inlining, template expansion, and link-time optimization obscure direct correspondence between source structure and machine code execution paths. Flame graphs therefore function as a bridge from low-level CPU cycle sampling to architecture-aware optimization decisions.

17.3.1 Sampling Model and Statistical Accuracy

Flame graphs are built from periodic sampling of the instruction pointer (IP). The sampling frequency must respect two constraints:

1. **Sufficient statistical representation** of execution hotspots.
2. **Non-intrusiveness** to avoid perturbing pipeline behavior.

Typical profiling settings:

```
$ perf record -F 999 -g -- ./app        # ~1 kHz sampling, call graph capture
```

Interpretation:

- Stacks where the IP frequently appears are *hot paths*.
- The *width* of a bar represents *proportional time spent*, not call count.

This sampling-based approach yields meaningful attribution without full tracing or instrumentation.

17.3.2 Collapsing Stacks into Aggregated Execution Paths

Raw perf output must be collapsed into aggregated stack traces:

```
$ perf script | stackcollapse-perf.pl > out.folded
```

A folded stack format contains each unique stack trace on one line, with a count representing its sampling frequency. For example:

```
main;process;compute;_ZN4math6mul@plt 12023
main;process;load_data;_ZNSt6vector19emplace_backERK 8301
```

This representation captures:

- Execution lineage (caller → callee relationships).
- Relative consumption of cycles across call chains.
- Shared subpaths aggregated to avoid duplicated visual patterns.

17.3.3 Flame Graph Rendering Model

Rendering:

```
$ flamegraph.pl --colors=java --width=1600 < out.folded > perf.svg
```

Interpretation rules:

- **X-axis**: aggregated time; wider boxes indicate higher contribution.
- **Y-axis**: call depth; each row represents one frame in the call chain.
- **Horizontal adjacency**: separate call-path contributions, not temporal sequence.

Important: flame graphs do **not** show *when* execution occurred—only *how often* a path contributed to total samples.

17.3.4 Mapping Optimized Code to High-Level Constructs

Due to inlining and template instantiation, the displayed function names often correspond to mangled or transformed symbols. Deconstruction follows ABI conventions (described in Part V):

```
$ c++filt _ZNK3foo4barIiEET_v
```

To preserve meaningful semantic identity:

- Use `-fno-omit-frame-pointer` to stabilize unwinding.
- Retain full DWARF types: `-g3 -ggdb`.
- Avoid symbol stripping during production builds intended for profiling.

Where necessary, flame graphs represent:

Symbol Type	**Resolution Strategy**
Inlined function	DWARF inline call site reconstruction.
Template instantiation	Use demangling + specialization context.
`std::` / container code	Enable libstdc++ pretty-printers (Chapter 16.5).

This allows attribution of performance costs back to specific C++ abstractions.

17.3.5 Cycle Attribution and Root-Cause Localization

Flame graphs identify *where* cycles accumulate, not *why*. Interpretation must pair flame graph results with microarchitectural stall attribution (Section 17.2):

Observation from Flame Graph	Next Diagnostic Step
Wide leaf function	Use `perf annotate` to inspect instruction scheduling.
Hot loops	Evaluate vectorization and ILP opportunities.
Heavy recursion	Investigate tail-call elimination opportunities.
`std::vector` / `std::map` hotspots	Check allocator behavior, capacity growth, and locality.

For example:

```
if a flame graph shows 40% time in std::string::append
→ check memory allocation reuse and SSO thresholds
→ measure `resource_stalls.sb` to confirm store saturation
```

Thus, flame graphs locate the *cost center*, while event grouping identifies the *structural bottleneck class* causing the cost.

17.3.6 Summary

Component	Purpose	Diagnostic Value
Stack sampling	Capture statistically meaningful execution paths	Low overhead; suitable for continuous profiling.

Component	Purpose	Diagnostic Value
Stack collapsing	Aggregate identical call chains	Shows true hotspot concentration and eliminates incidental variance.
Flame graph visualization	Represent hierarchical cost distribution	Allows intuitive navigation of performance-critical code.
Cycle attribution	Relate hotspots to architecture-level stall categories	Enables selecting the correct micro-architectural optimization.

Flame graphs are not a standalone analysis method; they are the **visual entry point** to pipeline diagnostics. Combined with perf stall classification and compiler-level IR inspection, they provide a complete workflow for performance engineering in optimized C++ systems.

17.4 Performance Bound Classification: Compute vs Memory vs Control

Optimizing a modern C++ system requires determining *why* execution is slow, not merely *where* instructions are executing. Microarchitectural performance is governed by three dominant bound classes:

- **Compute-bound** — performance limited by execution throughput or arithmetic intensity.
- **Memory-bound** — performance limited by data fetch latency or bandwidth.
- **Control-bound** — performance limited by branch prediction and speculative execution stability.

Correct classification is essential. Applying optimization strategies intended for the wrong class produces no measurable improvement. This section establishes a rigorous framework for distinguishing these bound categories using hardware counter analysis, flame graph inspection, and top-down performance modeling.

17.4.1 Compute-Bound Execution

A pipeline is **compute-bound** when arithmetic or instruction retirement rate is the limiting factor. The execution core is busy, but instruction-level parallelism (ILP) or SIMD utilization is insufficient to saturate available execution ports.
Key indicators:

Signal	Interpretation
High IPC (Instructions Per Cycle), near core peak	Indicates efficient backend utilization with minimal stall delays.
Port utilization skew	Suggests bottlenecked execution resources, often due to arithmetic specialization or insufficient ILP.
Low memory stall counters	Implies the working data set is cache-resident, reducing DRAM impact on performance.
Minimal misprediction overhead	Indicates stable and predictable control flow with accurate branch prediction.

Common C++ causes:

- Scalar loops with insufficient vectorization.
- Function-level inefficiencies hidden behind abstraction layers but preserved after optimization.
- Excessive precision usage (e.g., `double` where `float` is sufficient in hot loops).

Optimization strategies:

Approach	Effect
SIMD vectorization	Increases throughput by executing multiple operations per instruction cycle.
Loop unrolling	Improves ILP and reduces control-flow overhead.

Approach	Effect
Algorithmic restructuring	Reduces computational cost by selecting more efficient data structures or algorithms.
Specialization	Removes dynamic dispatch overhead and enables additional compiler optimization.

For compute-bound code, *adding more cores does not increase single-thread performance.*

17.4.2 Memory-Bound Execution

Workloads become **memory-bound** when waiting for data dominates execution time. Memory-bound latency shows up as widespread pipeline stalls despite available compute capacity.
Key indicators:

Metric	Meaning
High `mem_load_retired.*`	Load stalls caused by DRAM or LLC latency.
High `cycle_activity.stalls_l3_miss`	The memory hierarchy is the dominant bottleneck.
Low IPC (< 1.0)	Execution pipelines are underutilized due to stalls.
Flame graphs dominated by simple loop frames	Indicates data arrival latency rather than arithmetic bottleneck.

Common C++ triggers:

- Pointer-chasing data structures (`std::list`, `std::map`, intrusive trees).
- Non-contiguous memory access patterns (AoS instead of SoA).
- Repeated allocation without pooling (cache eviction of active structures).
- Containers resized frequently without capacity planning.

Optimization strategies:

Approach	**Effect**
Data layout refactoring (SoA, flattened trees, packed arrays)	Improves locality and reduces cache miss rate.
Prefetching or explicit cache warm-up	Hides memory latency by overlapping loads with computation.
Batching and tiling	Ensures computation operates within cache-friendly windows.
Reducing indirection layers	Decreases pointer chasing and improves cache efficiency.

For memory-bound workloads, algorithmic improvements often exceed micro-optimizations.

17.4.3 Control-Bound Execution

A workload becomes **control-bound** when branch prediction failures, speculative execution rollbacks, or unpredictable decision paths limit effective forward progress. Key indicators:

Metric	Interpretation
High `branch-misses / branches`	Predictor accuracy is low.
Non-linear flame graph structure	Indicates branching complexity and diverse code paths.
Frequent speculative flushes in `perf annotate`	Speculation errors are degrading throughput.
IPC drop proportional to mispredict penalty	Frontend stalls caused by pipeline recovery latency.

Common C++ causes:

- Data-dependent branches in tight loops.
- Polymorphic call sites without devirtualization.
- Poorly structured or deeply nested conditional logic.
- State machines with high branching entropy.

Optimization strategies:

Approach	Effect
Replace branches with arithmetic form (branchless programming)	Eliminates mispredicts entirely
Move dispatch tables from runtime computation to compile-time resolution (`constexpr`)	Reduces control entropy
Use static polymorphism / CRTP where feasible	Removes vtable dispatch

Approach	Effect
Reorder condition checks based on observed probability	Aids predictor training

Control-bound code often benefits from *predictability*, not raw computational strength.

17.4.4 Determining Bound Class: Diagnostic Workflow

A standardized pipeline for classification:

```
Step 1: Measure IPC, stall counters, and branch mispredict rate (perf stat -d).
Step 2: Generate flame graph to locate execution hotspots.
Step 3: Attribute stall source using top-down methodology:
        - If Retiring fraction low → slowdown is pipeline-bound.
        - If Backend bound + high memory stalls → memory-bound.
        - If Backend bound + high port pressure → compute-bound.
        - If Bad Speculation high → control-bound.
Step 4: Apply targeted optimization strategy based on class.
```

This workflow prevents inefficient optimization cycles and ensures correctness of performance decisions.

17.4.5 Summary

Correctly identifying whether a workload is compute-, memory-, or control-bound is the **foundation** of performance engineering. Every other optimization effort derives from this classification.

Bound Class	Primary Limiting Factor	Typical Signal	Correct Optimization Lever
Compute-bound	Execution throughput	High IPC, RS full	Vectorization, ILP, specialization
Memory-bound	Data supply latency	High L3/DRAM miss stalls	Data layout, locality optimization
Control-bound	Speculation accuracy	High branch mispredict rate	Branch elimination, predictor stabilization

17.5 Examples: Deriving Stall Source Percentages on Skylake

Stall attribution on modern Intel Skylake-class microarchitectures requires correlating sampled hardware events to structural performance categories. Skylake partitions execution behavior into three primary constraint domains:

- **Frontend Bound** — instruction supply / decode constraints.
- **Backend Bound** — execution resource contention or memory latency.
- **Bad Speculation** — branch misprediction and speculative execution rollback.

The Top-Down Microarchitectural Analysis methodology classifies pipeline slots into these categories and yields quantitative stall attribution percentages. This section demonstrates stall percentage derivation using `perf` event groups and interprets the results in the context of optimized C++ workloads.

17.5.1 Required perf Event Groups for Skylake

To acquire necessary counters for top-down analysis:

```
$ perf stat -e \
  cycles,instructions, \
  idq_uops_not_delivered.core, \
  uops_issued.any, \
  uops_executed.core, \
  resource_stalls.rob, \
  resource_stalls.rs, \
  resource_stalls.sb, \
  cycle_activity.stalls_l3_miss, \
  branch-misses,branches \
  -- ./app
```

These measurements capture:

Event	Stall Attribution Domain
`idq_uops_not_delivered.core`	Frontend supply inefficiency
`resource_stalls.rs`	Backend core execution congestion
`resource_stalls.rob`	Retirement bottleneck and dependency delays
`cycle_activity.stalls_l3_miss`	Memory latency stall severity
`branch-misses / branches`	Control-bound misprediction rate

17.5.2 Example Output from Real Execution

Example `perf stat` summary (abbreviated):

```
 4,200,000,000      cycles
```

```
 3,000,000,000      instructions              #   0.71 IPC
   180,000,000      branch-misses
 1,200,000,000      idq_uops_not_delivered.core
 1,800,000,000      resource_stalls.rs
   950,000,000      resource_stalls.rob
   410,000,000      cycle_activity.stalls_l3_miss
```

17.5.3 Computing Stall Domain Percentages

1. IPC and Retirement Efficiency

```
IPC = instructions / cycles = 3.0B / 4.2B   0.71
```

Versus Skylake theoretical retirement throughput 4 instructions per cycle:

```
Retiring Efficiency   0.71 / 4   18%
```

→ The pipeline is mostly stalled.

2. Bad Speculation (Control-Bound)

```
Branch Mispredict Rate = branch-misses / branches
Assume branches   1.1B → rate   180M / 1.1B   16%
```

16% misprediction rate is structurally high; speculative flush recovery likely contributes ~**10–20%** stall time.

3. Frontend Bound

```
Frontend Stall Share = idq_uops_not_delivered.core / cycles
 1.2B / 4.2B   28.6%
```

→ ~29% of total cycles are limited by instruction fetch/decode supply.

4. Backend Bound

Break backend into *core-bound* vs *memory-bound*:

- RS pressure → core execution congestion:

```
RS Stall Share = resource_stalls.rs / cycles   1.8B / 4.2B   42.8%
```

- ROB stall share suggests dependency constraints:

```
ROB Stall Share = 950M / 4.2B   22.6%
```

- Memory latency share:

```
Memory Stall Share = cycle_activity.stalls_l3_miss / cycles   410M / 4.2B   9.8%
```

Backend interpretation summary:

Subclass	Share	Meaning
Core Execution Bound	$\sim 43\%$	ALU / port pressure or mixed scalar workload
Dependency / Retirement Bound	$\sim 23\%$	Long dependency chains, partial vectorization
Memory Latency Bound	$\sim 10\%$	Data locality issues present but not dominant

17.5.4 Final Stall Attribution Breakdown

Bound Class	Percent of Cycles	Dominant Signal
Backend (Core)	$\sim 43\%$	`resource_stalls.rs`

Bound Class	Percent of Cycles	Dominant Signal
Frontend	$\sim 29\%$	`idq_uops_not_delivered.core`
Bad Speculation	$\sim 16\%$	Branch misprediction rate
Memory Latency	$\sim 10\%$	`cycle_activity.stalls_l3_miss`
Useful Work	$\sim 18\%$	IPC relative to peak retire width

This reveals:

- The workload is **backend execution bound**, not memory-bound.
- Frontend supply issues are secondary, but not negligible.
- Branch prediction contributes nontrivial waste but is not dominant.
- Memory subsystem behavior is acceptable; performance is *not* DRAM latency limited.

17.5.5 Interpretation and Optimization Direction

Given this profile, effective optimization strategies include:

Optimization Target	Rationale
Improve ILP / reduce dependency chains	Alleviate RS pressure and ROB stalls
Strengthen vectorization and type specialization	Reduce scalar ALU saturation

Optimization Target	Rationale
Consolidate instruction footprint / reduce inline bloat	Mitigate frontend starvation
Consider branchless forms where possible	Lower mispredict penalty footprint

Incorrect strategies to avoid:

- Memory prefetch tuning (problem is not memory-bound).
- Allocator changes (store buffer not under pressure).
- Thread parallelism for speedup (bottleneck is single-thread execution throughput).

17.5.6 Summary

Stall source derivation on Skylake requires:

1. Collecting microarchitectural counters.
2. Converting raw counts into *normalized stall shares*.
3. Classifying pipeline behavior into compute, memory, or control-bound categories.
4. Selecting optimizations aligned to the dominant structural bottleneck.

This quantifies performance in a way that directly maps to the compiler and architecture model described in prior chapters, enabling principled and repeatable optimization rather than trial-and-error tuning.

Part VIII

SYSTEM ENGINEERING CASE STUDIES (FULL STACK)

Chapter 18

Linux Kernel Compilation, Boot, and Live Debugging

18.1 Kernel Toolchain Integration

The Linux kernel is not built with the same compilation assumptions as user-space C++ binaries. The toolchain that produces the kernel must enforce architectural determinism, ABI stability, minimal runtime dependencies, and predictable code generation. Kernel compilation is therefore an explicit contract between:

- The **compiler** (GCC or Clang, configured in kernel mode),
- The **assembler** (binutils `as` or LLVM integrated assembler),
- The **linker** (`ld` from binutils or `lld` under restricted compatibility),
- The **C library boundary model** (the kernel cannot depend on glibc),
- The **bootloader and firmware environment**, which define the execution entry vector.

Understanding how these components integrate is essential before analyzing runtime debugging or kernel memory model behavior.

18.1.1 Kernel-Supported Compiler Feature Subset

The Linux kernel enforces a strict compiler capability contract:

- No exceptions (`-fno-exceptions`)
- No RTTI (`-fno-rtti`)
- No stack protector unless explicitly enabled via Kconfig (`CONFIG_CC_STACKPROTECTOR`)
- No use of the standard C++ library (kernel is written in C, with limited C++ allowed only in restricted environments)
- Reliance on compiler built-in intrinsics rather than libc-provided routines
- Fixed calling convention according to System V AMD64 ABI, but with additional constraints for interrupt/trap entry frames

The kernel build system validates compiler compatibility at configuration time:

```
$ make menuconfig
$ make CC=gcc
```

If GCC emits code requiring glibc or unwinder frames, the kernel build will fail. The kernel maintains its own lightweight runtime (atomic ops, memcpy, memset, division helpers) to avoid external linking.

18.1.2 Assembler and Linker Role in Kernel Layout

The kernel's ELF layout differs from user executables:

- PIE is **not** used; the kernel operates in a fixed virtual address map.
- `.text`, `.data`, and `.rodata` are placed into explicitly controlled memory regions.
- Special linker sections (`.init.*`, `.exit.*`, `.smp_locks`) govern initialization lifetime and hot/unhot code segmentation.

The kernel build system uses linker scripts such as `arch/x86/kernel/vmlinux.lds.S` to enforce:

```
SECTIONS
{
  . = KERNEL_BASE;
  .text : { *(.text*) }
  .rodata : { *(.rodata*) }
  .data : { *(.data*) }
  .bss  : { *(.bss*) }
}
```

This defines physical and virtual memory layout at boot.
The assembler (`as`) emits relocations that are resolved at *link* time, not runtime, because the kernel cannot rely on a dynamic loader (`ld.so` is user-space only).

18.1.3 Kernel ABI and Syscall Interface Boundaries

The kernel defines a stable ABI surface consisting of:

- System call interface (via `syscall` / `sysenter` / `int 0x80` paths),

- VDSO interfaces for fast time queries,
- io_uring, futex, and memory-map primitives.

The kernel **does not** guarantee ABI stability for any internal symbols. Only the syscall table and UAPI headers form the compatibility contract.
Toolchain impact:

- GCC must avoid optimizing across syscall boundary assumptions.
- Inline assembly blocks (`asm volatile("syscall" ...)`) must conform to the System V AMD64 calling convention.
- Clang/LLVM must reproduce exact register clobber semantics when used as kernel compiler.

18.1.4 Kernel Configuration and Build System (Kbuild)

Compilation flow:

```
$ make defconfig          # baseline configuration
$ make -j$(nproc)         # parallel kernel compilation
```

Kbuild orchestrates:

- Per-directory Makefile recursion,
- Dependency scanning for cross-architecture header selection,
- Unit-level compilation flags derived from architecture constraints (`arch/x86/Makefile`).

Every compilation unit may include architecture-specific flags:

```
KCFLAGS += -mno-sse -mno-red-zone -fno-stack-protector
```

No user-level runtime assumptions are permitted. The kernel cannot rely on red-zone space, because interrupt handlers may clobber stack beyond `rsp`.

18.1.5 Cross-Compilation and Toolchain Targeting

Kernel builds commonly target alternate architectures:

```
$ make ARCH=x86_64 CROSS_COMPILE=x86_64-linux-gnu-
```

The cross-compiler must provide:

Component	Requirement
`gcc` or `clang`	Supports kernel-compatible feature subset
`ld`	Correct relocation model for kernel virtual memory mapping
`objcopy`, `objdump`	Used to package boot images and symbol tables
`nm`	Used for internal dependency resolution

Version compatibility is strict; kernel releases encode minimum supported GCC/Clang versions in `Documentation/process/changes.rst`.

18.1.6 Summary

Component	Kernel Requirement	Toolchain Constraint
Compiler	No runtime library dependencies; fixed calling conventions	`-fno-exceptions -fno-rtti -mno-red-zone`
Assembler	Full control over section placement	`objtool` validation of frame correctness
Linker	Deterministic ELF layout, no dynamic relocation	Custom linker scripts control memory model
Runtime model	No glibc; kernel provides intrinsics internally	Must not generate external libc calls

Kernel toolchain integration is fundamentally a *whole-program compilation model*, not a dynamic linking model. Unlike user-space C++ binaries—where the compiler participates in a multi-stage dynamic loader pipeline—the kernel forms a closed system, with all symbol resolution fixed at link time and all code execution performed without external runtime support.

18.2 QEMU + GDB Step-Controlled Boot Path Analysis

Controlled kernel boot tracing allows precise observation of state transitions from firmware handoff through early kernel initialization. QEMU, when paired with GDB, provides a deterministic execution environment that reproduces architectural state transitions, interrupt enablement points, paging setup, and early stack initialization without requiring physical hardware access. This capability is essential for understanding the kernel's execution semantics as compiled by GCC.

18.2.1 QEMU Execution Environment as a Deterministic CPU Model

QEMU emulates x86-64 CPU microarchitectural behavior while preserving architectural correctness (register file, paging, segment descriptor interpretation, APIC state). It does not model speculative execution internals or dynamic ops scheduling. Therefore, analysis at this stage focuses on:

- Instruction sequence correctness,
- Control flow integrity,
- Memory access ordering (architecturally visible),
- Boot register and descriptor initialization.

Launching the kernel under QEMU with GDB stub enabled:

```
$ qemu-system-x86_64 \
    -kernel arch/x86/boot/bzImage \
```

```
    -append "console=ttyS0 nokaslr" \
    -nographic -s -S
```

Flags:

Flag	Meaning
`-S`	Halt CPU on reset before executing the first instruction
`-s`	Open a GDB remote debugging server at port 1234
`-nographic`	Use serial console only (disable graphical display)
`nokaslr`	Disable kernel ASLR to ensure stable and repeatable code layout

This produces a static, repeatable boot entry position for analysis.

18.2.2 Attaching GDB and Initial Execution Boundary

Attach GDB:

```
$ gdb vmlinux
(gdb) target remote :1234
```

`vmlinux` must be the unstripped ELF produced by kernel linking (not bzImage). It contains:

- Full symbol table,
- Debug DWARF,
- Accurate section layouts.

Set a break at the architecture-specific entry point:

```
(gdb) break start_cpu
(gdb) continue
```

For x86-64, early execution may initially be in 16-bit real mode entry or 32-bit trampoline code; however, QEMU virtual BIOS hands control to the decompressor stub, which ultimately transitions into 64-bit long mode before entering `start_kernel()`.

18.2.3 Stepping Through the Boot Decompression Phase

The kernel decompressor (`arch/x86/boot/compressed`) is compiled with the same compiler but under a restricted runtime environment:

- No paging initially,
- Flat real-mode or early protected-mode addressing,
- No stack preservation guarantees beyond controlled setup.

Use:

```
(gdb) layout asm
(gdb) stepi
```

Observe:

- GPR initialization in real mode,
- Transition to protected mode via CR0/CR4 writes,
- GDT installation,
- Entry to decompressed kernel image.

This stage verifies that GCC-generated code matches the expected environment constraints (no reliance on red-zone, no segment-relative assumptions early on).

18.2.4 Transition to `start_kernel()` and Subsystem Bring-Up

Once identity paging and long mode are established, execution proceeds to:

```
start_kernel()
  → setup_arch()
  → mm_init()
  → trap_init()
  → sched_init()
  → rest_init()
```

Setting conditional breakpoints:

```
(gdb) break start_kernel
(gdb) break setup_arch
(gdb) break early_idt_handler
```

This reveals:

- Interrupt table initialization correctness,
- Memory model boot-time identity mapping integrity,
- Kernel stack pointer installation before scheduler start.

Verification focuses on correctness of code generation under kernel flags (`-mno-red-zone`, struct alignment rules).

18.2.5 Dissection of Paging Setup and Virtual Memory Transition

A key boot verification step is inspecting the transition from identity mapping to full kernel virtual memory space at `PAGE_OFFSET`. Dump active page tables:

```
(gdb) monitor info mem
(gdb) x/32gx $cr3
```

Here, the goal is to ensure that relocations and linker script–defined segments align with the paging structure produced at runtime.
This confirms that:

- The kernel ELF layout matches the execution virtual memory model,
- `ld` and GCC-generated relocation assumptions are preserved.

18.2.6 Summary

Component	**Observed Through QEMU+GDB**	**Verification Objective**
Real-mode and trampoline code	Single-step execution	Confirm transition correctness
Decompressor and early protected mode	Instruction trace	Ensure stack/register model portability
Paging and long mode enable	CR0 / CR3 / CR4 / MSR state inspection	Validate memory identity assumptions
Scheduler and subsystem initialization	Symbol breakpoints	Confirm correct entry into process execution

Step-controlled kernel boot analysis enables deterministic validation that **GCC-generated kernel code satisfies architectural, calling convention, and memory initialization contracts**, all without reliance on hardware instrumentation facilities.

18.3 System Call Return Path Disassembly

The return path of a system call is the completion of the privileged-to-unprivileged execution transition. Unlike the entry sequence, which establishes kernel context, the return sequence must guarantee correct restoration of architectural state, privilege level, and observable memory effects. This return boundary defines the *completion semantics* of all Linux system calls. A correct understanding of this path ensures that any GCC-level optimizations within kernel code respect register, stack, and control-flow invariants across privilege transitions.

18.3.1 Return Path Overview

For x86-64, system call return transitions through the following layered sequence:

```
sys_call_table[nr]()
    ↓
do_syscall_64()
    ↓
entry_SYSCALL_64_tail
    ↓
sysretq   (fast-path)  or  iretq (slow-path / interrupts)
```

The return mechanism is determined by processor state and flags. The kernel must restore register values and privilege-level stack boundaries precisely. Any deviation results in immediate privilege or memory errors.

18.3.2 Tail Section: `entry_SYSCALL_64_tail`

Assembly excerpt (simplified):

```
entry_SYSCALL_64_tail:
```

```
    testq   $TS_COMPAT, %rcx
    jne     handle_compat_syscall_return

    movq    %rax, %rdi          # Return value already in %rax
    callq   syscall_return_slowpath

    jmp     restore_regs_and_sysretq
```

Register-level guarantees at this stage:

- `%rax` contains syscall return value or error code.
- No user-controlled state is restored until the transition instruction is executed.
- Kernel stack remains resident; no switching until privilege-level transition.

18.3.3 Fast vs Slow Return Paths

Two architectural paths exist:

Path	Instruction	Trigger Condition	Characteristics
Fast return	`sysretq`	No pending signals, no rescheduling, clean task flags	Preserves forward progress performance
Slow return	`iretq`	Signal handling required or return to different CPL state	More strict and fully restores segment and EFLAGS state

`sysretq` assumes:

- Canonical user-mode `%rip` and `%rsp` are already valid.

- RFLAGS-compatible model persists across boundary.
- No security-sensitive state requires full reinitialization.

If these conditions fail, `iretq` is selected; this guarantees full state restoration but incurs higher latency.

18.3.4 Stack and `pt_regs` Restoration

Kernel entry saved CPU state into a `pt_regs` structure located at the top of the task's kernel stack:

```
struct pt_regs {
    unsigned long r15, r14, r13, r12;
    unsigned long rbp, rbx, r11, r10;
    unsigned long r9, r8, rax, rcx;
    unsigned long rdx, rsi, rdi, orig_rax;
    unsigned long rip, cs, eflags, rsp, ss;
};
```

Return restoration:

```
popq    %r15
popq    %r14
popq    %r13
...
popq    %rdi
popq    %rbp
```

This model is strictly maintained; GCC's kernel compilation rules forbid frame pointer omission unless the unwinder graph is provably correct (validated by `objtool`).

18.3.5 Symbol Boundary Verification via Disassembly

Using GDB:

```
(gdb) break entry_SYSCALL_64_tail
(gdb) continue
(gdb) disassemble /m entry_SYSCALL_64_tail
(gdb) stepi
```

Then inspect the return instruction:

```
(gdb) x/i $rip
```

Expected output patterns:
Fast path:

```
sysretq
```

Slow path (signal pending, scheduling event, or traced process):

```
iretq
```

Verification objective:

- Ensure correct stack frame unwind sequence,
- Confirm that `%cs`, `%ss`, and RFLAGS return to user values,
- Validate that no stale kernel address leaks to user-mode.

18.3.6 Error Code Propagation and `-errno` Semantics

System call implementations return negative `errno` encodings:

- Kernel places `-EXXX` in `%rax`.
- Userspace wrappers in glibc translate to `errno` and return `-1`.

Verification disassembly confirms that no user-mode side-effects occur before the boundary instruction.
Example audit:

```
(gdb) print $rax
```

Return is applied *after* privilege exit, not before, ensuring exception visibility correctness.

18.3.7 Summary

Component	**Responsibility**	**Verified Through Disassembly**
`entry_SYSCALL_64_tail`	Pre-return state evaluation	Instruction sequencing and flag handling
`pt_regs` unwind	Restore architectural registers	Stack frame integrity
`sysretq` / `iretq`	Privilege boundary transition	Safety invariants and correct CPL resolution

Component	**Responsibility**	**Verified Through Disassembly**
Return value propagation	Kernel-to-user ABI contract	`%rax` validity and error code rules

The system call return path is the architectural mirror of the entry path: it completes the privilege transition while preserving execution correctness, security invariants, and ABI stability. Disassembly confirms that GCC-generated kernel code follows these obligations without deviation.

18.4 Page Table + Virtual Memory Initialization Walkthrough

The transition from early identity-mapped execution to the kernel's full virtual address space defines the core memory model under which all subsequent kernel subsystems execute. This phase establishes the initial page table hierarchy, maps the kernel text/data segments into canonical high-half virtual addresses, installs the direct physical memory map, and prepares the address space for scheduler activation and user-mode process creation. Correctness at this stage is mandatory: any inconsistency causes immediate triple faults or undefined behavior before debugging infrastructure is available.

18.4.1 Architectural Memory Model Baseline

On x86-64, virtual memory uses a 4-level or 5-level paging hierarchy, depending on hardware capabilities:

```
CR3 → PML4 → PDPT → PD → PT → Physical Page
```

Key invariants:

- All addresses used during initialization must be canonical.
- Instruction fetch and data access must reference mapped pages.
- Kernel segments must be aligned to page boundaries as enforced by linker script.

Linux uses the **higher-half kernel model**:

Region	Typical Virtual Base	Description
Kernel text/data	`ffffffff81000000`	Linked static kernel image
Direct physical map	`ffff888000000000`	Linear mapping of RAM frames
Per-CPU region	`fffffe0000000000`	CPU-local structures

The initial identity map is temporary and removed once full virtual memory is live.

18.4.2 Initial Page Table Creation (Early Boot)

The kernel decompressor constructs a minimal 64-bit bootstrap page table. After entering long mode, control transfers to the relocated kernel image, which initializes the full memory map in `setup_arch()`.
Critical function path:

```
start_kernel()
  → setup_arch()
     → early_alloc_pagetable()
     → paging_init()
```

`early_alloc_pagetable()`:

- Allocates initial top-level page directory.
- Maps kernel text and data using large (2 MiB) pages for TLB efficiency.
- Establishes temporary identity mapping enabling code execution continuity.

Representative code fragment (simplified conceptual form):

```
void __init paging_init(void) {
    init_top_pgt = alloc_pgt_page();
    map_kernel_text(init_top_pgt);
    map_kernel_rodata(init_top_pgt);
    map_kernel_data(init_top_pgt);
    map_phys_mem(init_top_pgt);   // direct physical mapping
    write_cr3(init_top_pgt);
}
```

Mappings use `PAGE_KERNEL` and `PAGE_KERNEL_EXEC` protection macros that expand to architecture-specific PTE flags.

18.4.3 Kernel Virtual Mapping: Text, Data, and BSS

The kernel linker script defines symbol boundaries:

```
VIRT_TEXT_START = 0xffffffff81000000;
VIRT_DATA_START = VIRT_TEXT_START + text_size;
```

These addresses are **compile-time constants** encoded into relocation fixups resolved by `ld` during kernel linking.
Disassembly verification:

```
(gdb) info files
...
0xffffffff81000000 - 0xffffffff81xxxxxx is .text
```

Page table entries for `.text` include the `NX` bit cleared, while `.rodata` pages have `NX` set and `RW` cleared.
Ensuring separation of executable and data memory regions enforces W^X policy in kernel mode.

18.4.4 Direct Physical Memory Map Construction

The direct mapping provides a 1:1 mapping of all physical RAM into a contiguous region in the virtual address space. This avoids repeated calls into architecture-specific translation logic when referencing physical frames.
Example calculation:

```
virtual = PAGE_OFFSET + physical
```

Here, `PAGE_OFFSET` is fixed (example `ffff888000000000`) and determined by architecture configuration.
This region supports:

- Slab and buddy allocator operations,
- Direct frame access for device drivers,
- Kernel crash dump introspection.

Large (2 MiB or 1 GiB) pages are preferred to reduce TLB pressure.

18.4.5 Page Attribute Enforcement and Memory Protection Flags

Relevant x86-64 PTE bit flags:

Flag	Meaning
`P` (Present)	Page is valid and mapped
`RW`	Writable

Flag	Meaning
`US`	User-accessible (kernel mappings generally clear this bit)
`NX`	Non-executable (if supported by CPU)
`PS`	Large page mapping (2 MiB or 1 GiB page size)
`PAT`	Cache type override / Page Attribute Table selection

During kernel mapping:

- User-access flags are cleared (`US=0`).
- NX is selectively enabled to enforce execute-only `.text`.
- Large pages are used where alignment and section granularity permit.

These constraints are preserved against GCC optimizations by explicit attribute annotations in architecture-specific headers, not by user-level compiler assumptions.

18.4.6 Debugging Page Table Initialization with QEMU + GDB

After kernel entry:

```
(gdb) break paging_init
(gdb) continue
(gdb) stepi
```

To inspect CR3:

```
(gdb) print/x $cr3
```

To walk page tables manually:

```
(gdb) x/8gx 0x<cr3_value>
```

To verify direct-mapped physical memory access:

```
(gdb) x/8gx 0xffff888000000000
```

Expected: deterministic mapping, no faults.

18.4.7 Summary

Component	Purpose	Verification Target
Bootstrap page table	Enable long mode	Identity-mapped execution correctness
Kernel high-half mapping	Execute kernel image	`.text` and `.data` protection flags
Direct physical map	Global RAM visibility	TLB locality and allocator efficiency
PTE flag enforcement	Memory safety	Correct W^X implementation
Debug tracing	Structural validation	Page hierarchy and `CR3` correctness

The kernel's virtual memory initialization is a contract between the GCC-generated ELF image, the linker script's symbolic layout, and architecture-enforced paging semantics. Correctness is demonstrated through disassembly and live introspection under a controlled emulator environment.

18.5 Examples: Stepping from `startup_64` into Scheduler Initialization

This section provides a controlled, instruction-level walkthrough of the execution path from the entry point of the kernel's 64-bit bootstrap code (`startup_64`) through early CPU bring-up and into the first activation of the scheduler. The objective is to verify that the GCC-compiled kernel image obeys architectural expectations regarding privilege level, stack initialization, paging setup, and context hand-off into `rest_init()`, which launches the idle task and the scheduler core.
The analysis uses QEMU with GDB attached, allowing repeatable breakpoints and controlled stepping.

18.5.1 Establishing Initial Debug Environment

Launch QEMU with debugging enabled:

```
$ qemu-system-x86_64 \
    -kernel arch/x86/boot/bzImage \
    -append "console=ttyS0 nokaslr" \
    -nographic -s -S
```

Attach GDB:

```
$ gdb vmlinux
(gdb) target remote :1234
```

You are now positioned at the CPU's reset halt, before executing the first instruction of the decompressor.

18.5.2 Breakpoint at `startup_64`

Set an initial breakpoint:

```
(gdb) break startup_64
(gdb) continue
```

`startup_64` performs:

- Establishment of early page tables,
- Control register setup (`CR0`, `CR4`, `EFER.LME`),
- Jump into the relocated kernel text.

Disassemble the entry block:

```
(gdb) disassemble /m startup_64
```

Expected major operations (simplified):

```
movq   initial_page_table, %cr3
movl   $MSR_EFER, %ecx
wrmsr
ljmp   $__KERNEL_CS, $entry_64
```

This transition enables full 64-bit mode and transfers to C-level initialization.

18.5.3 Transition to `start_kernel()`

After architectural setup, execution flows through:

```
startup_64
  → x86_64_start_kernel
     → start_kernel
```

Set breakpoint:

```
(gdb) break start_kernel
(gdb) continue
```

Verify call stack:

```
(gdb) bt
```

Expected top-level call:

```
start_kernel()
```

Confirm that:

- Paging is active,
- Kernel stack pointer is correct,
- Interrupts are disabled (checked via RFLAGS.IF bit).

18.5.4 Core Initialization Path into Scheduler Bring-Up

`start_kernel()` performs global subsystem setup, eventually reaching:

```
rest_init()
  → kernel_init()
     → sched_init()
```

Set breakpoints:

```
(gdb) break rest_init
(gdb) break sched_init
(gdb) continue
```

Disassemble:

```
(gdb) disassemble /m sched_init
```

Key scheduler initialization operations:

- Creation of the idle task (`init_task`),
- Initialization of per-CPU runqueue structures,
- Setup of load balancing state and topology domains.

Representative C-level excerpt (conceptual):

```
void __init sched_init(void) {
    init_idle(current, smp_processor_id());
    rq = cpu_rq(smp_processor_id());
    rq->curr = rq->idle;
}
```

This establishes the idle task as the first schedulable entity.

18.5.5 First Context Switch Activation

The scheduler is first invoked in:

```
rest_init()
  → schedule_preempt_disabled()
```

Set breakpoint:

```
(gdb) break schedule
(gdb) continue
```

Inspect register state prior to first task switch:

```
(gdb) info registers
```

Expected behavior:

- `current` points to idle task (`swapper`),
- Task state is `TASK_RUNNING`,
- Stack pointer resides in per-CPU kernel stack region.

The first `schedule()` call does not switch context—it validates the idle thread and returns immediately.

18.5.6 Summary

Execution Phase	**Verified Element**	**Key Diagnostic**
`startup_64`	CPU mode and paging initialization	Disassembly of MSR writes and `CR3` load
`start_kernel()`	System-wide initialization entry	Stack and segment state correctness
`rest_init()`	Transition to scheduler bring-up	First scheduling loop boundary
`sched_init()`	Idle task and runqueue setup	Validation of scheduler data structure initialization
First `schedule()`	Beginning of runtime scheduling control	Ensures no premature context switch

This walkthrough confirms that:

- The GCC-generated kernel follows the expected control transfer order,
- Page tables and virtual memory are valid before scheduler activation,
- Scheduler initialization is deterministic and architecture-compliant,
- Early execution does not depend on userspace ABI conventions.

At this point, the system is considered *operational*; multi-tasking and userspace process launch can proceed.

Chapter 19

Bare-Metal C++ Runtime Construction

19.1 Manual CRT (`crt0.s`) and ABI-Conformant Startup

Constructing a minimal C++ runtime requires replacing the standard C runtime initialization sequence normally provided by glibc's startup objects (`crt1.o`, `crti.o`, `crtn.o`). In fully controlled environments such as bare-metal kernels, unikernels, embedded systems, or research operating systems, this initialization must be implemented manually while still conforming to the System V AMD64 ABI. The objective is to provide the execution environment required for calling `main()`, ensuring defined register state, stack alignment, and proper termination semantics without relying on libc.

19.1.1 Architectural Requirements for Startup Code

When control is transferred to the program by the loader (whether a bootloader or a bare-metal entry vector), the runtime startup code must establish:

1. **Canonical stack pointer alignment** to **bytes**, mandated by the ABI before any function call.
2. **Program entry point linkage**, typically named `_start`, referenced by the ELF header.
3. **Argument and environment pointer capture**, passed to `main()` in a form compatible with the C++ ABI.
4. **Zero-initialized `.bss` region**, ensuring static object correctness.
5. **Relocation fixups** (for position-independent binaries, if applicable).
6. **C++ static initialization** sequencing prior to invoking `main()`.

No standard library calls are available at this stage. All operations must be implemented in pure assembly and minimal C.

19.1.2 Prototype Startup Assembly (`crt0.s`)

A minimal ELF ABI-compliant `_start` implementation for x86-64:

```
    .global _start
_start:
    mov %rsp, %rdi          # argc = initial stack pointer contents
    lea 8(%rsp), %rsi       # argv = next address after argc
```

```
    and $-16, %rsp          # Align stack to 16 bytes (ABI requirement)
    call __crt_init         # Perform runtime initialization (BSS, constructors)

    call main               # int main(int argc, char** argv)

    call __crt_fini         # Invoke destructors before exit

    mov %rax, %rdi          # Use main() return value as exit code
    mov $60, %rax           # SYS_exit
    syscall
```

Key rules:

- `_start` **must not** assume any preserved registers.
- `_start` must not rely on red-zone memory (kernel compilers disable red-zone; user-space must not rely on it during `_start` before stack alignment).
- The final termination must invoke system call `exit`, not return.

19.1.3 `__crt_init`: BSS Zeroing and Static Constructors

The `.bss` section contains zero-initialized objects. The linker exports boundary labels (not namespaced; defined in linker script):

```
extern uint8_t __bss_start;
extern uint8_t __bss_end;

static void __crt_init() {
    uint8_t* p = &__bss_start;
    while (p < &__bss_end) {
        *p++ = 0;
```

```
    }

    extern void (*__init_array_start[])(void);
    extern void (*__init_array_end[])(void);

    for (size_t i = 0; i < (__init_array_end - __init_array_start); ++i) {
        __init_array_start[i]();
    }
}
```

Constructor invocation sequences are defined by the **Itanium C++ ABI**, not invented per implementation:

- Global and static objects are registered in `.init_array` at link time.
- The startup code walks the function pointer array in definition order.

19.1.4 `__crt_fini`: Destructor Sequencing

Correct C++ runtime shutdown requires invoking destructors for global/static objects:

```
static void __crt_fini() {
    extern void (*__fini_array_start[])(void);
    extern void (*__fini_array_end[])(void);

    for (size_t i = (__fini_array_end - __fini_array_start); i > 0; --i) {
        __fini_array_start[i - 1]();
    }
}
```

Destructor ordering is reverse of initialization order, ensuring dependencies release in a valid lifetimes sequence.

19.1.5 ABI Conformance Rules That Must Be Preserved

Requirement	Specification Source	Enforcement
Stack alignment	System V AMD64 ABI	`_start` aligns stack before first `call`
Argument passing	System V AMD64 ABI register conventions	`argc` in `%rdi`, `argv` in `%rsi`
Static initialization ordering	Itanium C++ ABI `.init_array`	`__crt_init()` walks constructor table
Destruction ordering	Itanium C++ ABI `.fini_array`	`__crt_fini()` applies destructors in reverse order
Exit semantics	Linux Syscall ABI	`_start` must perform `syscall exit`, not return

Failure in any of these rules leads to:

- Undefined behavior in global object lifetimes,
- Incorrect function call boundaries,
- Stack misalignment causing crashes in vectorized function calls,
- Incomplete tear-down and resource leakage.

19.1.6 Summary

Building a manual CRT replaces the assumption of a hosted runtime environment with a fully deterministic, ABI-compliant execution bootstrap:

- `_start` provides the minimal architectural transition into C++.
- C++ static initialization and destruction are explicitly driven through `.init_array` and `.fini_array`.
- No external libc components are required.
- Semantic correctness is preserved by adhering to the System V AMD64 ABI and the Itanium C++ ABI.

This foundation enables construction of freestanding C++ runtimes for kernels, microcontrollers, hypervisors, and high-assurance embedded systems.

19.2 Eliminating glibc and Implementing Runtime Primitives

In a freestanding C++ environment, the standard C library and the standard C++ library are unavailable by definition. The compiler, however, still assumes the presence of certain runtime facilities unless explicitly overridden. Eliminating glibc therefore requires constructing a minimal support layer that satisfies GCC's internal lowering assumptions, supports essential C++ runtime semantics, and provides controlled access to the underlying execution environment (kernel, hypervisor, or bare metal). The resulting runtime must define symbol contracts, memory allocation primitives, and exception/termination semantics entirely within the program's own binary.

19.2.1 Hosted vs Freestanding: What the Compiler Expects

The C++ standard differentiates between:

- **Hosted environment**: full support of libc, libstdc++, and system calls.
- **Freestanding environment**: only core language features guaranteed; no standard library beyond minimal headers.

To compile in freestanding mode:

```
g++ -ffreestanding -fno-exceptions -fno-rtti -nostdlib -nostartfiles
```

Mandatory implications:

Feature	Provided by Compiler?	Must be Implemented Manually?
Integer arithmetic	Yes	No
Objects with static storage	Yes	Initialization required
Global constructors / destructors	Compiler emits constructor tables	Must call `.init_array` / `.fini_array`
Dynamic memory (`new`, `delete`)	Compiler may emit calls to allocation APIs	Must supply custom allocator
Exception unwinding	Compiler generates unwind tables, but requires runtime unwinder	Typically disabled in bare-metal (`-fno-exceptions`)

Thus, the runtime must provide **just enough** infrastructure to satisfy the compiler, not the entire libc.

19.2.2 Required Runtime Symbols

Even in freestanding mode, GCC may emit calls to certain symbols unless disabled:

- `memcpy`, `memset`, `memcmp`
- `memmove`
- `__stack_chk_fail` (if stack protector is not disabled)
- `__cxa_atexit` and `__cxa_finalize` (for static destructors, unless replaced)

To avoid external linkage:

```
-fno-stack-protector
-fno-exceptions
-fno-unwind-tables
-fno-asynchronous-unwind-tables
```

Custom replacements must ensure correctness and predictable performance:

```
extern "C" void *memcpy(void* dst, const void* src, size_t n) {
    unsigned char* d = static_cast<unsigned char*>(dst);
    const unsigned char* s = static_cast<const unsigned char*>(src);
    while (n--) *d++ = *s++;
    return dst;
}
```

Non-optimized versions are acceptable initially; optimized variants may later use architecture-specific intrinsics.

19.2.3 Implementing `new` and `delete`

The compiler expects the following symbols:

```
void* operator new(std::size_t size);
void operator delete(void* ptr) noexcept;
```

For bare-metal systems lacking virtual memory, the simplest allocator is a bump-pointer region:

```
static uint8_t heap[HEAP_SIZE];
static std::size_t offset = 0;

void* operator new(std::size_t size) {
    size = (size + alignof(std::max_align_t) - 1) & ~(alignof(std::max_align_t)-1);
    if (offset + size > HEAP_SIZE) return nullptr;
```

```
    void* p = &heap[offset];
    offset += size;
    return p;
}

void operator delete(void*, std::size_t) noexcept {}
```

In kernel or embedded scenarios, this allocator would later be replaced by slab, buddy, TLSF, or custom region allocators.

19.2.4 Avoiding glibc for System Interaction

In a bare-metal configuration, no system call interface exists. The environment defines the lowest-level I/O mechanism:

- UART registers (SoC firmware),
- Memory-mapped device controllers,
- Supervisor-mode hypercalls,
- BIOS/UEFI stubs,
- `syscall` only if a kernel is present.

For example, writing to a memory-mapped UART:

```
static volatile uint8_t* const UART_TX = reinterpret_cast<uint8_t*>(0x10000000);

inline void crt_putc(char c) {
    *UART_TX = static_cast<uint8_t>(c);
}
```

Higher abstractions (print routines, logging subsystems, formatted output) are layered atop these primitives, not libc.

19.2.5 Termination Semantics Without `exit()`

Final teardown must not assume libc finalization. In hosted Linux environments, termination uses `SYS_exit`:

```
mov %rax, %rdi    # exit code -> rdi
mov $60, %rax     # SYS_exit
syscall
```

On bare-metal systems, termination may instead:

- Halt the CPU,
- Trigger a machine reset,
- Trap to firmware,
- Or loop indefinitely.

19.2.6 Summary

Removing glibc requires rebuilding the minimal runtime surface that C++ compilation assumes:

Runtime Component	**Implementation Strategy**
Startup	Manual CRT entry via `_start`
Global init/fini	Traverse `.init_array` and `.fini_array` sections
Memory	Provide custom `new`/`delete` or region allocator
String / memory ops	Implement core routines such as `memcpy`, `memset`, etc.

Runtime Component	**Implementation Strategy**
Termination	Use `syscall exit` or platform-specific halt mechanism
System services	Replace libc wrappers with explicit system or hardware interface calls

This runtime provides a deterministic, self-contained execution substrate suitable for embedded systems, research kernels, secure execution domains, hypervisors, and high-integrity environments where reliance on external standard libraries is unacceptable.

19.3 Console Output + Interrupts + Minimal Heap

A freestanding C++ runtime must supply three foundational subsystems to enable structured program execution: (1) a deterministic output interface for diagnostic visibility, (2) a minimal interrupt dispatch layer to handle external or timer-generated events, and (3) a controlled heap allocator to support dynamic storage. These subsystems must operate independently of libc, libstdc++, and kernel services. Their design is constrained entirely by the execution environment's hardware model and by the System V AMD64 ABI.

19.3.1 Console Output: Direct Hardware or MMIO Write Path

Without glibc, output cannot rely on `printf()` or file descriptors. Console output must directly write to a hardware-defined sink. On bare-metal x86-64 platforms two models are typical:

1. **Memory-Mapped Framebuffer (e.g., text mode VGA)**
   Writes directly to the display memory region.

2. **Memory-Mapped UART**
   Writes character output to a serial port.

Example: MMIO UART transmit register access:

```
static volatile uint8_t* const UART_TX = reinterpret_cast<uint8_t*>(0x10000000);

inline void crt_putc(char c) {
    *UART_TX = static_cast<uint8_t>(c);
}
```

Minimal line-buffered output:

```
void crt_print(const char* s) {
    while (*s) {
        if (*s == '\n') crt_putc('\r');
        crt_putc(*s++);
    }
}
```

This provides deterministic output timing and does not require formatting logic. More advanced formatting is layered atop this primitive; no variadic printf parsing is required initially.

19.3.2 Interrupt Descriptor Table (IDT) and Interrupt Gate Setup

Bare-metal execution must handle traps, exceptions, and optionally timer interrupts. This requires defining the **Interrupt Descriptor Table**, loading it with the `lidt` instruction, and associating each interrupt vector with an entry stub that performs:

- Register state preservation,
- Transition to a C-level interrupt handler,
- Restoration and `iretq`.

Example interrupt stub (simplified):

```
.global isr_timer
isr_timer:
    pushq %r15; pushq %r14; pushq %r13; pushq %r12;
    pushq %r11; pushq %r10; pushq %r9;  pushq %r8;
    pushq %rsi; pushq %rdi; pushq %rbp; pushq %rbx;
    pushq %rdx; pushq %rcx; pushq %rax;
```

```
    call timer_interrupt_handler

    popq %rax; popq %rcx; popq %rdx; popq %rbx;
    popq %rbp; popq %rdi; popq %rsi; popq %r8;
    popq %r9;  popq %r10; popq %r11; popq %r12;
    popq %r13; popq %r14; popq %r15;

    iretq
```

C++ side:

```
extern "C" void timer_interrupt_handler() {
    // Acknowledge interrupt source, update internal timing state.
}
```

This preserves full calling convention correctness, allowing C++ code to run inside an interrupt context without violating register state.

19.3.3 Interrupt Controller Initialization

Hardware interrupt routing must be enabled manually. Example cases:

- **APIC / x2APIC** on x86-64, configured via MSRs.
- **PLIC / GIC** on RISC-V or ARM systems.

The runtime must:

1. Initialize the interrupt controller,
2. Unmask required interrupt vectors,
3. Ensure that interrupt return (`iretq`) is valid.

Interrupt latency and reentrancy behavior must be controlled; the minimal runtime **does not** provide preemption or scheduling unless explicitly designed.

19.3.4 Minimal Heap and Allocation Strategy

A freestanding C++ environment cannot assume a virtual memory-backed heap. The allocator must treat memory as a finite region with no external expansion. The simplest correct model is a **bump-pointer allocator**, where memory grows monotonically:

```
static constexpr std::size_t HEAP_SIZE = 64 << 10;
static std::aligned_storage_t<HEAP_SIZE, alignof(std::max_align_t)> heap_storage;
static std::size_t heap_offset = 0;

void* operator new(std::size_t size) {
    size = (size + alignof(std::max_align_t) - 1) & ~(alignof(std::max_align_t)-1);
    if (heap_offset + size > HEAP_SIZE) return nullptr;
    void* p = reinterpret_cast<uint8_t*>(&heap_storage) + heap_offset;
    heap_offset += size;
    return p;
}

void operator delete(void*) noexcept {}
```

Characteristics:

Property	Behavior
Allocation	O(1), monotonic
Free operation	No-op
Fragmentation	None
Lifetime model	Permanent (until system reset)

This is sufficient for early runtime, logging buffers, static objects, and message queues. More advanced runtimes may replace this allocator with region-based recycling or TLSF-based O(1) allocators once interrupts and timers are stable.

19.3.5 Summary

The minimal functional runtime stack consists of:

Component	Responsibility	Key Constraint
Console Output	Deterministic debug visibility	Direct MMIO / port access; no `stdio`
Interrupt System	Controlled event handling	ABI-safe register preservation
Minimal Heap	Enable dynamic storage use	No external memory manager dependency

This layer enables C++ to run in a deterministic, platform-controlled environment without libc, syscalls, or operating system support. It establishes the foundation upon which higher-level abstractions such as cooperative scheduling, device drivers, and message-passing subsystems can be constructed.

19.4 Static Constructors Without Runtime Support

In a bare-metal C++ environment, global and static objects must still be constructed before `main()` executes, even though no standard C runtime or dynamic loader is available to perform this task. The compiler and linker continue to emit constructor metadata in `.init_array`, but it is the responsibility of the manually implemented runtime to *invoke* these constructors in the correct order, without interfering with ABI rules or violating memory initialization guarantees. Failure to handle constructor sequencing results in uninitialized global state, undefined object lifetimes, or incorrect ordering dependencies in complex subsystems.

19.4.1 How GCC Represents Static Initialization

Under the Itanium C++ ABI (used by GCC on x86-64), the compiler translates each global or namespace-scope object with a non-trivial constructor into an entry in the **`.init_array`** section.
The linker concatenates all such entries into a contiguous region:

```
[ __init_array_start ]
   ctor_0
   ctor_1
   ...
   ctor_n
[ __init_array_end ]
```

Each entry is a *function pointer* of type:

```
using ctor_t = void (*)();
```

No implicit ordering guarantees are imposed beyond link-unit concatenation; however, link order is deterministic and stable across reproducible builds.

19.4.2 Constructing the `.init_array` Region Manually

The runtime must call each constructor exactly once, before invoking `main()`:

```
extern "C" void __crt_run_constructors() {
    extern ctor_t __init_array_start[];
    extern ctor_t __init_array_end[];

    for (ctor_t* f = __init_array_start; f != __init_array_end; ++f) {
        (*f)();   // Invoke constructor
    }
}
```

This function must be invoked in `_start` (or `__crt_init`) *after* `.bss` has been zero-initialized and before `main()` is called. Calling constructors before `.bss` initialization produces invalid runtime state.

19.4.3 Destruction Without a Runtime: `.fini_array`

Destructor order is the reverse of constructor order, as required by the ABI to ensure dependent static objects unwind correctly.

```
extern "C" void __crt_run_destructors() {
    extern ctor_t __fini_array_start[];
    extern ctor_t __fini_array_end[];

    for (ctor_t* f = __fini_array_end; f != __fini_array_start; ) {
        (--f)[0]();   // Reverse iteration
    }
}
```

If the runtime does not call destructors, global objects with RAII-managed resources (buffers, device handles, log pipes) will leak or fail to release hardware state consistently.

19.4.4 Aligning Constructor Execution With Memory Model Constraints

Global lifetime rules require the following invariants:

1. **`.bss` must be fully zeroed** before any constructor runs.
2. **No heap allocation** should occur during construction unless `operator new` is already available.
3. **Interrupts must be disabled**, to avoid concurrent access before system initialization completes.
4. **The console device must be operational**, or objects producing debug output will fail silently.

A typical ordering discipline:

```
_start
  → Clear BSS
  → Initialize heap (optional)
  → Enable console output
  → __crt_run_constructors()
  → main()
```

19.4.5 Common Failure Cases and Their Root Causes

Failure Mode	Cause	Correction
Static object not constructed	`.init_array` not invoked	Ensure runtime calls constructor table
Double construction	Runtime invoked `.init_array` twice	Invoke only once before `main()`
Destructors never run	No `.fini_array` invocation	Call destructor table after `main()` or before halt
Heap used before allocator init	Constructors allocate storage	Initialize minimal allocator **before** constructor pass
Deadlock in constructors	Constructors assume interrupts enabled	Defer enabling interrupts until after `main()`

Constructors must be considered *pure initialization code* and executed in a deterministic isolated environment.

19.4.6 Summary

Global/static initialization remains fully supported in freestanding C++—but **only if triggered explicitly by the runtime**. GCC emits constructor metadata automatically, but no initialization occurs unless the runtime:

1. Defines `_start`,
2. Clears `.bss`,
3. Walks `.init_array` manually,
4. Optionally invokes `.fini_array` at termination.

This design preserves C++ language guarantees without relying on glibc, libstdc++, or dynamic loader machinery, enabling C++ to function predictably in kernel, hypervisor, embedded, and secure execution domains.

19.5 Examples: Booting a C++ ELF Directly Under QEMU

This section demonstrates how to construct, link, and execute a freestanding C++ binary as a *bootable image* under QEMU, without an operating system, dynamic loader, libc, or firmware runtime. The objective is to verify that the manually defined runtime startup sequence (`_start`, `.bss` zeroing, `.init_array` constructor calls) is valid and that the ELF image is structurally compatible with the CPU reset execution model once positioned as a kernel entry payload.
This illustrates the *end-to-end viability* of the bare-metal runtime implemented in Chapter 19.

19.5.1 Minimal Linker Script for Bare-Metal ELF

A custom linker script defines:

- The executable load address,
- Segment layout,
- Symbol exposure for `.init_array` and `.bss`.

Example (`linker.ld`):

```
ENTRY(_start)

SECTIONS
{
  . = 0x100000;                /* Physical load address */

  .text ALIGN(4K) :
```

```
{
  *(.text .text.*)
}

.rodata ALIGN(4K) :
{
  *(.rodata .rodata.*)
}

.data ALIGN(4K) :
{
  *(.data .data.*)
}

.bss ALIGN(4K) :
{
  __bss_start = .;
  *(.bss .bss.* COMMON);
  __bss_end = .;
}

.init_array ALIGN(8) :
{
  __init_array_start = .;
  *(.init_array*)
  __init_array_end = .;
}

.fini_array ALIGN(8) :
{
  __fini_array_start = .;
  *(.fini_array*)
  __fini_array_end = .;
```

```
  }
}
```

This script ensures identity-mapped access and page-aligned segment boundaries, matching standard bootloader expectations.

19.5.2 Minimal Bootable C++ Program

```
extern "C" void crt_putc(char c);

static int counter = 0;

struct Demo {
    Demo() { crt_putc('I'); crt_putc('N'); crt_putc('I'); crt_putc('T'); }
} demo_instance;

extern "C" int main() {
    while (true) {
        crt_putc('0' + (counter++ % 10));
    }
}
```

This example confirms:

- `.init_array` constructor execution (`Demo()` runs before `main()`),
- `.bss` was zero-initialized (`counter == 0` initially),
- Output path is functioning (`crt_putc` must be operational).

19.5.3 Startup Assembly (`crt0.s`)

```
.global _start
```

```
_start:
    mov %rsp, %rdi                  # argc (unused)
    and $-16, %rsp                  # ABI-required alignment

    call __crt_zero_bss
    call __crt_run_constructors
    call main

hang:
    hlt
    jmp hang
```

This matches the ABI rules already established:

- Stack alignment,
- Constructor invocation,
- No return from `main()`.

19.5.4 Building the ELF

```
g++ -ffreestanding -fno-exceptions -fno-rtti -mno-red-zone \
    -nostdlib -nostartfiles -Wl,-T,linker.ld \
    crt0.s runtime.cpp main.cpp -o kernel.elf
```

Flags:

Flag	Meaning
`-ffreestanding`	Disable assumptions about hosted environments (no standard runtime guaranteed).

Flag	Meaning
`-mno-red-zone`	Avoid using the red-zone so stack remains valid across interrupts and traps.
`-nostdlib`	Exclude default C/C++ runtime libraries such as `libc` and `libstdc++`.
`-T linker.ld`	Use a custom linker script to define memory layout rather than the default.

19.5.5 Running the ELF Under QEMU

Direct ELF boot (QEMU treats `kernel.elf` as kernel entry payload):

```
$ qemu-system-x86_64 \
    -kernel kernel.elf \
    -nographic
```

If the UART output function maps to QEMU's emulated `-serial stdio`:

```
IN IT01234567890123456789...
```

This verifies:

- The binary executed without loader involvement.
- Global constructors executed before `main()`.
- The `.bss` region started zeroed.
- The console write primitive is functional.

19.5.6 Debugging the Boot Sequence

Enable stop-on-reset and GDB stub:

```
$ qemu-system-x86_64 -kernel kernel.elf -s -S -nographic
```

Attach debugger:

```
$ gdb kernel.elf
(gdb) target remote :1234
(gdb) break _start
(gdb) continue
(gdb) stepi
```

Inspect `.init_array` execution:

```
(gdb) break __crt_run_constructors
```

Inspect `.bss`:

```
(gdb) print __bss_start
(gdb) x/32bx __bss_start
```

Expected: all zeroed.

19.5.7 Summary

This example confirms that:

Runtime Component	**Verified Behavior**
ELF layout	Correct and bootable without external loader
Constructor execution	`.init_array` walked correctly

Runtime Component	**Verified Behavior**
Memory model	`.bss` zero initialization validated
Console output	Hardware primitive operational
Main execution	Program control reached and sustained

This constitutes a *minimal, fully self-sufficient, freestanding C++ execution environment*, suitable for:

- kernel prototyping,
- embedded firmware,
- hypervisor and enclave runtimes,
- OS research.

Chapter 20

High-Performance C++ Systems Optimization Project

20.1 Devirtualization → Inlining → Vectorization Pipeline

High-performance C++ systems rely on the compiler's ability to eliminate abstraction overhead while preserving correctness. Modern GCC performs a staged optimization pipeline in which **dynamic dispatch is reduced**, **call boundaries are collapsed**, and **loop bodies are restructured** for SIMD execution. This section describes how devirtualization enables inlining, which in turn enables vectorization. These steps are not independent; they form a **dependency chain** that determines whether high-level code can be lowered to hardware-efficient machine instructions.

20.1.1 Precondition: Alias, Escape, and Type Visibility

Before GCC attempts to optimize a polymorphic call, it evaluates whether:

1. **The concrete dynamic type is statically discoverable**, or at least sufficiently constrained.
2. **Pointer aliasing does not obscure object identity**, i.e., pointer provenance is known.
3. **The object does not escape to unknown callers**, preventing assumptions about replacement.

These properties are determined during GIMPLE-SSA construction, where the compiler performs:

- Range and provenance analysis,
- Escape analysis,
- De-facto class hierarchy resolution.

If the compiler can prove the object's dynamic type is unique or non-overridden, **virtual dispatch can be replaced** with a direct function call.

20.1.2 Devirtualization: From Virtual Call to Direct Call

Given:

```
struct Base { virtual int f(int) const; };
struct Derived : Base { int f(int) const override; };

void run(const Base* p, int x) {
    int r = p->f(x);
}
```

If `run()` is compiled with sufficient information (LTO or visible translation unit), and the compiler proves `p` always refers to `Derived`, the call:

```
p->f(x)
```

becomes:

```
Derived::f(x)
```

This removes:

- VTable load,
- Indirect branch,
- Pointer misprediction hazard.

The result is:

- Lower instruction count,
- Reduced branch misprediction,
- Better pipeline predictability.

Devirtualization is a structural enabler for inlining.

20.1.3 Inlining: Eliminating Call Boundaries

Once devirtualized, the function becomes eligible for inlining, provided it meets the inliner cost model:

- Function body size $<$ inlining threshold,

- No excessive register pressure,
- ABI boundary is internal to the compilation unit,
- Call is in a hot path determined by PGO (if available).

Inlining effects:

Result	**Benefit**
Call frame eliminated	No prologue/epilogue overhead
Arguments become SSA values	Enables constant propagation
Control flow merges	Enables loop fusion and simplification
Memory accesses can be hoisted	Improves locality and vectorizability

Inlining moves the computation into context where *semantic structure* becomes optimizable.

20.1.4 Vectorization: SIMD Lowering After Structural Simplification

The vectorizer operates on **loops with clear induction structure** and **uniform side effect patterns**. After inlining removes function boundaries:

- Loops become single, analyzable regions.
- Scalar expressions form canonical recurrence relations in SSA.
- The compiler identifies opportunities for:
  - Packed arithmetic (XMM/YMM registers),

- Load/store grouping,
- Branch elimination into mask operations.

Example pre-vectorization:

```
for (size_t i = 0; i < n; ++i)
    out[i] = a[i] * scale + bias;
```

After successful devirtualization and inlining:

- The loop body reduces to a pure arithmetic lambda on arrays.
- The vectorizer emits AVX2 or AVX-512 instructions depending on compile flags:

```
vmovaps ymm0, [rdi + rax*4]
vmulps  ymm0, ymm0, ymm3
vaddps  ymm0, ymm0, ymm4
vmovaps [rsi + rax*4], ymm0
```

No conditional branches, no indirect calls, no runtime indirection remain.

20.1.5 Practical Optimization Implications

Optimization Stage	Trigger Condition	Visibility Requirement
Devirtualization	Type uniqueness proven	Class hierarchy information must be visible (LTO recommended)
Inlining	Cost model threshold satisfied	Call site and function definition must be visible

Optimization Stage	Trigger Condition	Visibility Requirement
Vectorization	Canonical loop, pure arithmetic, and aligned access patterns	No aliasing ambiguity and predictable memory strides

Failing earlier steps **prevents** later steps entirely.
For example, failing to devirtualize prevents inlining, which prevents loop simplification, which prevents vectorization.
This pipeline is therefore **strictly sequential**.

20.1.6 Summary

The pipeline:

```
Virtual Call
    ↓ (Type Proven)
Devirtualized Call
    ↓ (Inlining Cost Pass)
Inlined Function Body
    ↓ (Loop Simplification and SSA canonicalization)
Vectorized Loop
```

Key insight:

> **High-performance C++ does not come from "writing low-level code," but from writing code whose structure enables the compiler to *lower* abstractions safely.**

Modern GCC, when given full program visibility and optimization freedom, can convert expressive object-oriented C++ into **tight SIMD-optimized code paths** equivalent to manually written assembly.

20.2 Memory Layout Re-Factoring for Cache Residency

High-performance C++ execution is fundamentally constrained by memory system behavior. Modern x86-64 cores can execute multiple arithmetic operations per cycle, but memory latency—particularly beyond L1—dominates execution cost. Effective optimization therefore requires restructuring data layouts and traversal order to maximize **cache residency**, **spatial locality**, and **predictable access stride**. This section focuses on memory layout refactoring, not algorithmic changes: the computation remains identical, but its representation in memory is reorganized to match the behavior of the CPU cache hierarchy.

20.2.1 Architectural Background: Latency and Bandwidth Constraints

Typical approximate memory access costs on contemporary x86-64 (e.g., Skylake, Zen2):

Memory Level	Latency (cycles)	Bandwidth Characteristics
Register File	1	Internal execution width-limited
L1 Data Cache	~ 4	64B per cycle load/store peak
L2 Cache	~ 12	Intermediate buffering layer
L3 Cache	~ 40–70	Shared across cores; non-uniform
DRAM	~ 120–300	Orders of magnitude slower

If working sets exceed L1/L2 residency, execution becomes **memory bound**, not

compute bound—even if vectorized.
This shifts optimization emphasis from instruction transformations to **data layout constraints**.

20.2.2 Array-of-Structs (AoS) vs Struct-of-Arrays (SoA)

Consider:

```
struct Particle {
    float x, y, z;
    float vx, vy, vz;
};

std::vector<Particle> P;
```

AoS layout:

```
[x y z vx vy vz] [x y z vx vy vz] ...
```

When computing `x += vx`, only two fields are used, but the cache must fetch all fields.
Cache bandwidth is wasted.
Refactoring to SoA:

```
struct Particles {
    std::vector<float> x, y, z;
    std::vector<float> vx, vy, vz;
};
```

Now:

- Loads of `x` and `vx` are contiguous,
- Vectorization applies directly,

- Working set reduces to active components.

This improves:

Property	AoS	SoA
Spatial Locality	Poor (unused fields intermixed)	High (relevant values contiguous)
SIMD Utilization	Low (strided gather)	High (contiguous loads)
Cache Residency	Unpredictable	Predictable and tunable

This restructuring is often the difference between **scalar** and **AVX-saturated** execution.

20.2.3 Aligning Data for SIMD and Line Size

Cache lines are 64 bytes. SIMD loads require alignment to avoid penalties and fallback to partial load paths.
For `float` arrays (4 bytes):

- 64 bytes / 4 bytes = 16 elements per cache line.
- AVX (256-bit) load fetches 8 floats at once.
- AVX-512 loads fetch 16 floats at once.

To allow optimal load and store grouping:

```
alignas(64) std::vector<float> x;
```

Or static arrays:

```
alignas(64) float x[N];
```

Alignment ensures:

- Fewer cross-line loads,
- No extra load micro-ops,
- Reduced TLB pressure due to larger effective stride uniformity.

20.2.4 Minimizing Working Set Size Through Compaction

Unnecessary per-object or per-element metadata increases footprint. Reducing structure size reduces cache miss rate.
Example: Instead of storing full state in each object, extract constants shared across many objects:

```
// Before
struct Node { float weight; float bias; };

// After
struct Node { float weight; };
float bias_global;
```

This compresses state and improves packing density.
For containers, prefer:

- `std::vector` over `std::list` (contiguous vs pointer-chasing)
- Manual memory pools over scattered allocation
- Page-aligned clustered pools for task-locality groups

20.2.5 Traversal Strategy and Prefetch-Favoring Order

Loops must follow **sequential, unit-stride** memory traversal to enable automatic hardware prefetch:

```
for (size_t i = 0; i < n; ++i) {
    x[i] += vx[i];
}
```

Avoid:

```
for (auto idx : random_index_list) {
    x[idx] += vx[idx];
}
```

Random access degrades to DRAM-bound latency.
If non-contiguous traversal is unavoidable, **software prefetch** can mitigate:

```
__builtin_prefetch(&x[i + PREFETCH_DISTANCE]);
```

Prefetch distance must match measured memory latency in cycles.

20.2.6 Summary

Effective memory layout optimization proceeds in the following transformation sequence:

```
High-Level Object Design
        ↓
Remove Unused or Duplicate State
        ↓
Convert AoS → SoA where possible
        ↓
```

```
Enforce 64B Alignment & Contiguous Allocation
        ↓
Rewrite Loop Traversal to Sequential Stride
        ↓
Measure → Adjust Working Set to Fit L1/L2
```

Resulting Performance Characteristics:

Capability	**Enabled By**
SIMD auto-vectorization	SoA + contiguous layout
Hardware prefetch success	Sequential access stride
Low latency compute loops	Working set fits in L1/L2
Stable scaling across cores	Reduced shared L3 contention

In high-performance C++, *data layout is performance*. Compiler optimizations are only effective when the underlying memory representation allows coherent, dense, predictable data movement.

20.3 PGO + LTO Combined Execution Optimization

Profile-Guided Optimization (PGO) and Link-Time Optimization (LTO) are orthogonal but complementary optimization strategies in GCC. When applied together, they enable the compiler to make globally informed decisions regarding inlining, branch prediction, indirect call elimination, and memory layout. The result is performance that cannot be attained with static heuristic optimization alone.
This section examines the combined PGO+LTO pipeline, the execution information it leverages, and the precise structural transformations that occur.

20.3.1 Rationale: Static Heuristics vs Profiled Behavior

Without profiling data, GCC must rely on static estimates:

- Branch probabilities inferred from code structure,
- Polymorphic call site assumptions,
- Loop iteration count guesses,
- Indirect call frequency heuristics.

Such heuristics approximate worst-case distributions and are inherently conservative. **Profile-Guided Optimization replaces assumptions with measured execution data**, allowing the optimizer to:

- Promote hot call paths to inline candidates,
- Devirtualize polymorphic dispatch based on observed dynamic type frequencies,
- Inline allocation and memory access patterns based on actual working-set usage.

20.3.2 The Two-Phase PGO Workflow

Phase 1: Instrumentation Build

```
g++ -O2 -fprofile-generate -flto -o app_profiled app.cpp ...
```

Executing the program generates `.gcda` counters capturing:

- Branch hit ratios,
- Loop iteration histograms,
- Call target frequency distributions.

Phase 2: Feedback-Driven Rebuild

```
g++ -O3 -fprofile-use -flto -o app_optimized app.cpp ...
```

LTO ensures that this runtime profile data is visible across all translation units, enabling cross-module inlining and global call graph restructuring.

20.3.3 Internal Optimization Effects

With PGO and LTO active, GCC's midend performs:

Optimization	Triggering Data	Effect
Branch Probability Adjustment	Relative branch execution counts	Reorders basic blocks to reduce pipeline stalls and improve fetch/decode efficiency.

Optimization	Triggering Data	Effect
Indirect Call Promotion	Call target frequency tables	Replaces virtual or indirect calls with direct calls in hot call sites, reducing dispatch overhead.
Cross-Module Inlining	Whole-program view under LTO	Inlines hot or small functions across translation units to eliminate call overhead.
Hot/Cold Code Partitioning	Execution density maps	Moves infrequently executed blocks into separate code sections to improve I-cache locality.
Loop Transformation Biasing	Loop iteration profile	Applies selective unrolling, fusion, or vectorization only where beneficial to execution throughput.

The key mechanism is **informed cost modeling**: transformations are only performed where the measured benefits exceed register pressure and memory expansion costs.

20.3.4 Example: Virtual Dispatch Collapse Under PGO

Consider:

```
struct Base { virtual float f(float) const = 0; };
struct DerivedA : Base { float f(float x) const override { return x * 2; } };
struct DerivedB : Base { float f(float x) const override { return x * 3; } };

float compute(const Base* p, float x) {
    return p->f(x);
}
```

With no profiling, GCC preserves virtual dispatch.
If runtime profile indicates `p` is `DerivedA` in >99% of calls:

- GCC rewrites dispatch to a **direct call** to `DerivedA::f`,
- Inserts a fallback indirect dispatch only for the rare path.

This eliminates:

- VTable loads,
- Indirect branch misprediction,
- Potential pipeline flushes.

20.3.5 Example: Cross TU Inlining Through LTO

Without LTO:

```
// a.cpp
float g(float);

// b.cpp
float compute(float x) { return g(x) + 1; }
```

`g(x)` cannot be inlined unless the programmer manually includes its definition.
With LTO:

- The full call graph is available at link time,
- `compute(x)` can be fully inlined and vectorized,
- The optimizer can eliminate redundant loads or recomputation.

This globally enables transformations such as:

- Constant propagation across translation units,
- Dead code elimination spanning modules,
- Layout merging for cold segments.

20.3.6 Combined PGO + LTO Optimization Model

```
Runtime Profile Collection
            ↓
Global Call Graph + Cost Model at Link Time (LTO)
            ↓
Profile-Driven Devirtualization and Inlining (PGO)
            ↓
Loop, Vectorization, and Memory Layout Optimization
```

The pipeline does **not** merely speed up functions; it **restructures the entire program** execution behavior around empirical runtime patterns.

20.3.7 Summary

Property	**Without PGO/LTO**	**With PGO + LTO**
Inlining Scope	Local (per translation unit)	Global (whole-program visibility)
Branch Prediction	Heuristic and static guesswork	Driven by measured runtime branch frequencies

Property	**Without PGO/LTO**	**With PGO + LTO**
Virtual Dispatch	Preserved at runtime	Eliminated or devirtualized in hot code paths
Loop Optimization	Pattern-based optimizations only	Profile-guided selective unrolling and vectorization
Code Layout	Arbitrary relative placement	Hot/cold partitioning to improve I-cache locality

In high-performance C++ systems, PGO and LTO together convert runtime dynamics into structural optimization decisions, enabling architecture-level saturation (vector, cache, and pipeline) that static compilation cannot achieve.

20.4 ABI Stability Under Optimized Transformations

High-performance optimization must preserve the Application Binary Interface (ABI) contract. While the optimizer may transform control flow, inlining boundaries, data layout access paths, and calling frequency, it **must not alter** externally observable calling conventions, name mangling rules, exception propagation semantics, or symbol visibility. This section clarifies which compiler transformations are ABI-neutral, which are ABI-sensitive, and how GCC enforces stability guarantees in the presence of aggressive optimization including LTO, PGO, vectorization, and devirtualization.

20.4.1 ABI Elements That Must Not Change

The System V AMD64 ABI defines binary interoperability rules across shared libraries, dynamic loaders, and user code. The following elements are invariant:

1. **Function Calling Conventions**
    - Argument and return value registers (RDI, RSI, RDX, RCX, R8, R9; XMM0+ for floating and vector types).
    - Stack alignment requirements (16-byte alignment at call boundaries).
    - Caller vs callee-saved register responsibilities.
2. **Object Layout for Standard Layout and Polymorphic Types**
    - Base class subobject offsets.
    - VTable pointer placement (typically first word of dynamic objects).
    - Typeinfo object identity.
3. **Mangling and Symbol Naming (Itanium C++ ABI)**

- Guarantees cross-language and cross-module link compatibility.

4. **Exception Unwind Encoding (DWARF CFI + LSDA Tables)**

    - Interface to stack unwinder must remain structurally valid under optimization.

These constraints define what the optimizer is allowed to modify **internally** without altering program linkage behavior.

20.4.2 Transformations That Are ABI-Neutral

The following classes of optimization do not change externally visible binary contracts:

Transformation	**ABI Impact**	**Reason**
Inlining (within a module)	None	Call site is replaced locally; no external linkage contracts are changed.
SSA and GIMPLE restructuring	None	These affect only internal compiler IR forms and do not alter linkage or symbol boundaries.
Loop transformations (unrolling, vectorization)	None	Control flow and iteration structure change, but public interfaces remain unchanged.
Constant and range propagation	None	The optimization influences values, not layout or external type signatures.

Transformation	ABI Impact	Reason
Code motion (LICM, hoisting)	None	Execution semantics are preserved without affecting symbol visibility or calling conventions.

These optimizations are always permissible during PGO and LTO.

20.4.3 Transformations That Are ABI-Sensitive

Certain optimizations must be guarded by symbol visibility and linkage rules:

Transformation	Risk	Control Condition
Devirtualization of external polymorphic calls	Assumes derived type identity at runtime.	Requires whole-program visibility, LTO, or explicit control of RTTI boundaries.
Cross-module inlining with LTO	May inline symbols that are not stable public API or reveal internal implementation details.	Safe only when both caller and callee are compiled and linked under the same LTO pipeline.
Structure layout re-packing	May change binary layout and break ABI compatibility.	Permitted only for types not shared or exposed across module or shared-library boundaries.

In general, optimizations are constrained by **linkage visibility**:

- `hidden` and `internal` symbols may be freely restructured.
- `default` visibility symbols must preserve layout and calling semantics.

20.4.4 Compiler and Linker Coordination Under LTO

During LTO, the compiler performs whole-program analysis. However, ABI stability is preserved by marking:

- Exported symbols as **non-relocatable** by layout,
- External calls as **devirtualization-blocked** unless type provenance is proven,
- VTable and RTTI structures as **identity-protected** objects.

Internally, GCC's LTO orchestration layer annotates:

```
// Marked internal TU-local function, eligible for inlining/elim.
__attribute__((visibility("hidden")))
static float process(float x);
```

vs

```
// Exported API function - ABI contract must remain stable.
__attribute__((visibility("default")))
float api_entry(float x);
```

This separation ensures that the optimizer may fully restructure internal computation while preserving binary boundary correctness.

20.4.5 Example: ABI-Preserving Devirtualization in Hot Contexts

Given:

```
struct Base { virtual double f(double) const; };
struct Derived : Base { double f(double x) const override { return x + 1; } };

extern void consume(const Base&);
```

If profiling determines `consume` is always invoked with `Derived`, GCC may perform:

- Direct call substitution **inside the TU**,
- But must **not alter the VTable** or remove Base's virtual function slot.

Generated code:

```
; Inside hot path
call Derived::f(double)

; But symbol table still exposes:
_ZN4Base1fEd:
    jmp *vtable(Base)+offset
```

The ABI-visible dispatch structure remains intact.

20.4.6 Summary

ABI stability is a **first-class optimization constraint**. GCC permits internal structural optimization while ensuring that:

- Module boundaries retain expected symbol forms,

- Exported type and function identities remain stable,
- Runtime linkability and exception unwinding remain correct.

In high-performance system development, the implementation must be aggressively optimized, but the **binary agreement surface must remain invariant**. This balance ensures scalability, interoperability, and long-term maintainability of optimized C++ systems.

20.5 Examples: Before/After Disassembly + perf Comparison Trace

This section illustrates the measurable behavioral consequences of PGO+LTO-driven optimization, devirtualization, and vectorization. The objective is not to show syntactic differences in source but to demonstrate **instruction-level structural evolution** and the subsequent microarchitectural performance impact.
A scalar loop is used as the baseline. It is deliberately memory-linear and branch-free to isolate changes introduced by the optimizer, rather than algorithmic reformulation.

20.5.1 Baseline Code (Unprofiled, No LTO)

```
double sum(const double* __restrict a, std::size_t n)
{
    double s = 0.0;
    for (std::size_t i = 0; i < n; ++i)
        s += a[i];
    return s;
}
```

Compile baseline:

```
g++ -O2 -fno-tree-vectorize -o baseline sum.cpp
```

Relevant disassembly (trimmed for clarity):

```
.L2:
    movsd  (%rdi,%rax,8), %xmm1
    addsd  %xmm1, %xmm0
    inc    %rax
    cmp    %rax, %rsi
    jne    .L2
```

Performance (`perf stat ./baseline` on Skylake-class core, n 10^8):

```
Instructions Retired: ~4.1e8
Cycles              : ~4.9e8
IPC                 : ~0.84
Estimated BW        : ~12.8 GB/s (memory-bound)
```

The loop is scalar and tightly coupled to dependency latency (`addsd` has a 3–5 cycle dependent chain delay).

20.5.2 Optimized Build (PGO + LTO + Vectorization)

Profile and optimize:

```
g++ -O3 -march=skylake -fprofile-generate -flto -o sum_prof sum.cpp
./sum_prof                              # runtime profiling
g++ -O3 -march=skylake -fprofile-use -flto -o sum_opt sum.cpp
```

Optimized disassembly (inner loop):

```
.Lhotvector:
    vmovupd (%rdi,%rax,8), %ymm1
    vaddpd %ymm1, %ymm0, %ymm0
    add    $4, %rax
    cmp    %rax, %rsi
    jb     .Lhotvector
```

Scalar tail cleanup omitted.
Key structural differences:

Aspect	Baseline	Optimized
ISA width	SSE scalar	AVX2 256-bit
Data per iteration	1 element	4 elements
Loop-carried dependency	Yes (serial)	No (parallel lanes)
Branch frequency	1 per element	1 per 4 elements
Vector loads alignment	Unaligned but contiguous	Same, but promoted to wider loads

20.5.3 Performance Result

```
perf stat ./sum_opt
```

Typical metrics:

```
Instructions Retired: ~1.4e8
Cycles           : ~2.2e8
IPC              : ~1.63
Estimated BW     : ~28-35 GB/s (approaching L2→L1 streaming limits)
```

Observations:

- IPC nearly doubled due to reduced dependency chaining and improved port utilization.
- Retired instructions decreased by ~65%.
- Memory throughput doubled, approaching front-end sustainable throughput.
- The loop transitions from latency-bound to bandwidth-bound.

20.5.4 Microarchitectural Reasoning

Phenomenon	Explanation
Dependency chain removal	AVX vector lanes compute in parallel, eliminating the scalar accumulation latency chain.
Load-Use penalty reduction	Wider loads reduce per-element addressing overhead and amortize memory latency.
Instruction retirement pressure	Reduced loop control overhead, as the branch executes far less frequently (75
Backend utilization	`vaddpd` instructions issue across multiple vector execution ports, efficiently saturating ALU throughput.

This transformation moves execution from **front-end loop overhead constraints** to **data feed limits**, aligning performance with architectural maximums.

20.5.5 Symbol and ABI Boundary Stability

Despite radical restructuring **inside** the function body:

- Symbol name remains unchanged (`_Z3sumPKdm` or equivalent under Itanium ABI).
- Calling convention is preserved (arguments and return still in registers as defined).
- No external code references are altered.

- The function remains link-compatible across dynamic/shared library boundaries.

Optimization affected **implementation**, not **interface**.

20.5.6 Summary

This example validates the core principle of modern C++ systems performance engineering:

> **Correctness is defined at the ABI boundary; performance is determined inside the boundary.**

The optimizer may restructure loops, apply vectorization, modify instruction scheduling, and change memory-access granularity — provided the externally visible ABI contract remains invariant.

Appedices

Appendix A - System V AMD64 ABI Reference

This appendix consolidates the binary-interface rules that govern C++ binaries produced by GCC for Linux x86-64. It is written as a **verification reference**: concise tables, exact register/stack rules, and minimal examples you can correlate with disassembly, DWARF, and linker views. The content reflects post-2020 practice on modern GCC and x86-64 cores (including AVX/AVX2/AVX-512 register files).

A.1 Integer & Floating-Point Register Classification

A.1.1 General-purpose (GP) registers

Register	Role	Volatility
RAX	return value, scratch	caller-saved
RBX	callee frame/temps	**callee-saved**
RCX	arg4, scratch	caller-saved
RDX	arg3, scratch	caller-saved
RSI	arg2	caller-saved

Register	Role	Volatility
RDI	arg1	caller-saved
RBP	frame pointer (optional)	**callee-saved**
RSP	stack pointer	special (must be restored by callee)
R8–R11	args5–8, scratch	caller-saved
R12–R15	callee temps	**callee-saved**

Notes:

- Integer args 1..6 map to: **RDI, RSI, RDX, RCX, R8, R9** (then spill to stack).
- `__int128` uses **RDX:RAX** for returns when not memory-classed.

A.1.2 Vector/FP registers (SSE/AVX/AVX-512)

Register class	Coverage	Role	Volatility
XMM0–XMM7	128-bit	FP/vector args/returns	caller-saved
XMM8–XMM15	128-bit	additional FP args	caller-saved
YMM/ZMM	256/512-bit	width aliases of XMM	caller-saved
XMM16–XMM31	(if supported)	extra FP/vector	caller-saved

Notes:

- Callee must **not** assume preservation of any XMM/YMM/ZMM; they are caller-saved under SysV.

- x87 stack regs (`st(0)`, …) are used only for `long double` (80-bit) and its complex variants.

A.2 Function Argument Mapping & "Shadow" Areas

A.2.1 Core mapping (fixed-arity functions)

1. **GP integer/pointer args** → RDI, RSI, RDX, RCX, R8, R9, then stack (right-to-left layout, 8-byte aligned).
2. **FP/scalar vector args** → XMM0–XMM7 (then XMM8–XMM15 if available), then stack.
3. Mixed aggregates follow the **SysV AMD64 classification** (A.3).

A.2.2 No Windows "shadow space"

- **SysV AMD64 has no 32-byte "shadow space."**
- The ABI defines a **128-byte red-zone** below RSP for leaf functions (A.4), but it is unrelated to Windows shadow space.

A.2.3 Variadic call support areas

`va_list` layout (SysV AMD64) is:

```
typedef struct {
  unsigned int gp_offset;      // bytes consumed from GP area
  unsigned int fp_offset;      // bytes consumed from FP (SSE) area
  void *overflow_arg_area;     // stack args beyond register arrays
  void *reg_save_area;         // home buffer for GP/FP register args
} __va_list_tag[1];
```

- For **variadic callees**, the prologue materializes a **register save area** so `va_arg` can fetch both GP and FP arguments consistently, regardless of where the caller passed them.

A.3 Return Value Encoding (scalar, aggregate, vector)

A.3.1 Scalars

Type	Return location
`int`, `long`, pointer	**RAX**
`__int128`	**RDX:RAX**
`float`, `double`	**XMM0**
`long double` (80-bit)	**x87 st(0)**
`_Complex float`, `_Complex double`	**XMM0:XMM1**
`_Complex long double`	**st(0):st(1)**

A.3.2 Aggregates & the SysV classification algorithm

- Every aggregate is split into 8-byte chunks and each chunk is classified as: **INTEGER**, **SSE**, **SSEUP**, **X87**, **X87UP**, **COMPLEX_X87**, or **MEMORY**.
- If any chunk is **MEMORY** (or the aggregate $> 2\times8$ bytes without a legal register assignment), the result is returned **via hidden sret pointer** passed in **RDI** (callee writes to that address).
- Otherwise up to two 8-byte chunks are returned in registers:
  - **INTEGER** chunks $\rightarrow$ RAX, then RDX

– **SSE/SSEUP** chunks → XMM0, then XMM1

Practical rules of thumb:

- **16 bytes** POD aggregates often return in regs if composed solely of INTEGER/SSE classes.
- Mixed integer/FP fields can still be register-returned if classification permits.
- Non-trivial C++ objects with user-defined copy/move typically use **sret**.

A.4 Stack Frame, Alignment & Red-Zone

A.4.1 Alignment rule

- The **caller** must ensure **RSP is 16-byte aligned *immediately before* `call`**. After the `call` pushes the 8-byte return address, the callee's frame has `(RSP+8) % 16 == 0`.

A.4.2 Prologue/Epilogue (canonical)

```
push   %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
mov    %rsp, %rbp
.cfi_def_cfa_register %rbp
sub    $N, %rsp          # align local area to 16B if needed
...
leave
.cfi_def_cfa %rsp, 8
ret
```

A.4.3 Red-zone (leaf function scratch)

- A **128-byte red-zone** exists **below** the current RSP which leaf functions may use without adjusting RSP.
- **Do not rely on the red-zone** in:
  - Kernel code (`-mno-red-zone`)
  - Signal/interrupt handlers
  - Code that may be probed by stack-walking tools or instrumentation

A.5 Variadic Functions & Homogeneous Aggregates

A.5.1 Variadic argument retrieval

- FP args to variadic functions are still passed in **XMM** regs when available; `va_arg(double)` reads them from the **FP portion** of `reg_save_area` using `fp_offset`, falling back to `overflow_arg_area` (stack) when exhausted.
- Mixing integer and FP varargs is fully supported; **ordering is preserved** by the `va_list` offsets.

A.5.2 "Homogeneous aggregates"

- The SysV AMD64 ABI does **not** define the AArch64-style "homogeneous aggregate" term as a separate rule.
- Practically, vectors such as `__m128`, `__m256`, `__m512` and aggregates composed **entirely** of the same FP/vector class are classified into **SSE/SSEUP** and passed/returned via XMM/YMM/ZMM according to the general classification in **A.3**.

A.6 DWARF Unwind Directives & Exception Frames

A.6.1 CFI in assembly (minimal, exception-safe)

```
.globl foo
.type  foo,@function
foo:
  .cfi_startproc
  push  %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset %rbp, -16
  mov   %rsp,%rbp
  .cfi_def_cfa_register %rbp
  sub   $32,%rsp
  ...
  leave
  .cfi_def_cfa %rsp, 8
  ret
  .cfi_endproc
.size foo, .-foo
```

Key CFI ops:

- **`.cfi_startproc` / `.cfi_endproc`**: mark frame scope.
- **`.cfi_def_cfa`**: define Canonical Frame Address (CFA).
- **`.cfi_offset %rbp, -16`**: tell unwinder where callee-saved regs are spilled relative to CFA.
- **`.cfi_def_cfa_register %rbp`**: switch CFA base to RBP after prologue.

A.6.2 C++ exceptions (Itanium model on ELF)

- Personality routine: **`__gxx_personality_v0`** (drives stack unwind and landing pads).
- Tables:
  - **`.eh_frame`** / **`.eh_frame_hdr`**: unwind CFI
  - **`.gcc_except_table`**: LSDA (landing pad/action table) per function with `try/catch`.
- Optimizations must maintain valid CFI for any function that can participate in unwinding, even under inlining and tail-merge. GCC/objtool validate this for kernel-like builds; for userspace, ensure you do not strip `.eh_frame` when exceptions are enabled.

A.7 Quick Reference Tables

A.7.1 Integer argument order

```
RDI → RSI → RDX → RCX → R8 → R9 → [stack…]
```

A.7.2 FP/vector argument order

```
XMM0 → XMM1 → … → XMM7 (→ XMM8..XMM15 if available) → [stack…]
```

A.7.3 Callee-saved set

`RBX, RBP, R12, R13, R14, R15` (and **RSP** must be restored).
All XMM/YMM/ZMM are **caller-saved**.

A.7.4 Return registers

- Integer/pointer: **RAX** (with **RDX** as high part if 128-bit)
- FP/vector scalars: **XMM0**
- Complex FP: **XMM0:XMM1**
- 80-bit `long double`: **st(0)**
- Small aggregates: **RAX/RDX** and/or **XMM0/XMM1** per classification; else **sret**.

A.8 Practical Verification Checklist

1. **Call sites**: Is **RSP** 16-byte aligned before `call`?
2. **Prologue CFI**: Are `.cfi_*` matching actual spills/moves?
3. **Red-zone usage**: Safe context (userspace leaf, no signals) or disabled?
4. **Small aggregate returns**: Do disassembly and classification agree (register vs sret)?
5. **Variadic access**: Is `va_list` initialized (gp/fp offsets) and reg save area present?
6. **Vector width**: Are FP args/returns in **XMM** regs and preserved only by caller?
7. **Callee-saved**: Are RBX/RBP/R12–R15 restored along all exits (including exceptions)?

Purpose Recap

These rules allow you to:

- **Audit GCC output** for ABI correctness,
- **Hand-write or patch** prologues/epilogues safely,
- **Design JIT trampolines** and interpose at PLT/GOT with confidence,
- **Correlate** DWARF CFI with actual frame layout for robust exception unwinding.

This appendix is intentionally compact; it is meant to be kept open beside `objdump -d`, `readelf -Ws`, and GDB while performing low-level diagnostics or constructing custom runtimes.

Appendix B - GCC Diagnostic and Dump Infrastructure

GCC exposes internal compilation stages through structured dump outputs. These facilities allow the full lowering path—from C++ source to GIMPLE, SSA, RTL, and final scheduled assembly—to be examined with precision. For system-level compilation, runtime ABI inspection, and performance tuning, the ability to correlate transformations at each stage is essential. This appendix provides the diagnostic workflow required to evaluate correctness and verify optimization behavior, particularly under aggressive inlining and link-time optimization.

B.1 Dump Invocation and Output Structure

Dump options are attached to the compile command and emit internal representations into structured files alongside the object output.
General format:

```
g++ -O2 -fdump-tree-all -c file.cpp
g++ -O2 -fdump-rtl-all -c file.cpp
```

Generated files follow:

```
file.cpp.XXXX
```

where `XXXX` denotes the specific transformation stage.

Recommended workflow

1. Enable tree dumps during semantic lowering and early optimization.
2. Enable SSA dumps to analyze value propagation and dominance structure.
3. Enable RTL dumps to inspect machine abstraction and register allocation.
4. Compare dumps before and after enabling vectorization, LTO, and PGO.

B.2 Tree Dumps and GIMPLE Phase Map

`-fdump-tree-*` captures transformations during the high-level and mid-level phases.
Key checkpoints:

Dump Name	Phase	Purpose
`original`	Post parsing, pre-lowering	Confirms correct semantic structure.
`gimple`	Canonical control/data flow	Baseline for all midend analysis.
`optimized`	After high-level passes	Shows inlining, constant propagation, dead code removal.
`vect`	After vectorization passes	Indicates feasibility and lane structure.
`inline`	Inlining decisions	Shows call-site elimination and cost feedback.

Example invocation:

```
g++ -O3 -fdump-tree-optimized -fdump-tree-vect -c compute.cpp
```

Interpretation principle:

- **GIMPLE is the representation to evaluate program logic independent of machine architecture.**
- Inlining, virtual devirtualization, loop canonicalization, and type-based alias analysis all occur here.

B.3 SSA Dumps and Value Flow Visibility

SSA form is central to all value-propagation decisions. Dumps:

```
g++ -O3 -fdump-tree-ssa -fdump-tree-dom -fdump-tree-fre -c file.cpp
```

Key analysis constructs:

- **SSA names** represent distinct value definitions.
- **Phi functions** appear at dominance frontiers and indicate control-dependent flow merges.
- **Range propagation** signals the optimizer's derived value constraints.

Researchers should inspect SSA for:

1. Eliminated loads/stores.
2. Inferred constant ranges on induction variables.
3. Hardening decisions (bounds checks preserved or removed).

B.4 RTL Dumps and Target-Lower Boundaries

`-fdump-rtl-*` documents the transition from GIMPLE to the machine-level IR that models registers, addressing modes, and instruction semantics.
Representative checkpoints:

Dump Name	Information
`expand`	First RTL form; GIMPLE lowered but unoptimized.
`cse1/cse2`	Common subexpression elimination at RTL.
`sched`	Post-scheduling with pipeline ordering.
`reload`	Register allocation and spill decisions.
`final`	Emit-ready instruction stream.

Command example:

```
g++ -O3 -fdump-rtl-expand -fdump-rtl-final -c kernel.cpp
```

Key interpretation focus:

- Instruction selection and addressing mode legalization.
- Register allocation pressure and spill cost structure.
- Scheduling alignment with microarchitectural ports.

B.5 Optimization Feedback: Inlining and Vectorization

GCC provides direct textual reasoning for specific optimizations:

```
g++ -O3 -fopt-info-inline -fopt-info-vec -c loop.cpp
```

Example (interpreted):

```
loop.cpp:17: note: loop vectorized with 4 lanes (cost model: fast)
loop.cpp:3: note: inlined function compute(): call frequency high, size small
```

These diagnostics confirm whether the compiler validated a transformation's cost-benefit model rather than merely attempting the transformation heuristically.

B.6 LTO Metadata and Whole-Program Visibility

When using link-time optimization, dump analysis must include link-stage IR:

```
g++ -O3 -flto -fopt-info -c module.cpp
g++ -O3 -flto -fopt-info -o app module.o other.o
```

Inspection commands:

```
gcc-nm --plugin=liblto_plugin.so app
gcc-objdump --plugin=liblto_plugin.so -dr app
```

LTO maintains:

- Complete cross-translation-unit call graph,
- Accurate devirtualization and alias analysis visibility,
- Consistent inlining decisions aligned with whole-program cost models.

B.7 Practical Procedure for Transformation Verification

To evaluate optimization correctness:

1. Dump **GIMPLE** and verify logical structure is preserved.
2. Dump **SSA** and confirm value propagation matches expected dominance trees.
3. Dump **RTL expand** and ensure memory access semantics and ABI are correct.
4. Dump **RTL final** and correlate assembly with register binding and scheduling.
5. Use `perf` to confirm pipeline saturation or stall-resolution behavior.

This progression ensures that **semantic correctness, structural transformation, and microarchitectural efficiency** are all validated consistently.

Purpose Restated

This diagnostic infrastructure:

- Makes the compiler pipeline **observable**,
- Enables **proof-of-correctness** for optimizations,
- Supports **research-level analysis** of compilation strategies,
- Provides stable workflows for **performance forensics** and **binary validation**.

It is the essential toolset for advancing from *using GCC* to *actively analyzing and controlling its compilation behavior*.

Appendix C - GDB, objdump, readelf, and perf Integration

This appendix establishes a unified methodology for correlating **symbolic program structure**, **compiler-transformed machine code**, and **runtime microarchitectural performance behavior**. The workflow is designed for verifying compiler output, diagnosing execution bottlenecks, and reconstructing semantic meaning from binary code. The methodology applies particularly to optimized C++ programs compiled under `-O3`, `-flto`, and PGO conditions.

C.1 Tracing Execution from `_start` to `main`

The entry point `_start` is provided by the runtime CRT object (`crt1.o`). It sets up the initial process state and invokes `__libc_start_call_main`, which ultimately transfers control to `main`.

Disassembly of the startup sequence

```
objdump -d --demangle --no-show-raw-insn a.out | less
```

Navigate to `_start`:

```
_start:
    xor    %ebp,%ebp
    mov    %rdx,%r9
    ...
    call   __libc_start_call_main
```

To confirm the transition chain:

```
gdb ./a.out
(gdb) starti
(gdb) si          # step instruction-by-instruction
(gdb) b main
(gdb) continue
```

This sequence verifies correct linkage of stack, argc/argv, TLS, and constructor invocation order.

C.2 Reconstructing Logical Structure from Disassembly

Identifying hot loops

```
objdump -dr --disassemble=compute a.out
```

Look for **loop-carried dependency chains**, induction variables, and memory stride patterns.

Mapping back to GIMPLE

Use previously generated dumps (`-fdump-tree-optimized` and `-fdump-tree-vect`) to correlate:

GIMPLE Construct	Assembly Indicator
`phi` node at loop head	register reload at loop boundary
Induction step (`i = i + 1`)	`inc` or `add $1` against loop index register
Vector lane parallelism	VEX-prefixed `vaddpd`, `vmulpd`, etc.

The reconstruction aligns **machine execution behavior** with **compiler-level representation**.

C.3 GOT/PLT Resolution Analysis

Dynamic linking introduces indirection via the Procedure Linkage Table (PLT) and Global Offset Table (GOT). To observe:

```
readelf -r a.out       # Relocation entries
readelf -Ws a.out      # Symbol table
objdump -d a.out | less
```

Runtime binding behavior can be observed in GDB:

```
(gdb) catch syscall open
(gdb) break *plt_function_entry
```

Objective:

- Determine if symbol bindings occur **eagerly** (`BIND_NOW`) or **lazy** via PLT stubs.
- Identify interposition and relocation deferral behavior.

C.4 Performance Counter Acquisition via perf

Baseline performance characteristics

```
perf stat ./a.out
```

Reports:

- Instruction-retired count
- Cycle count
- IPC (Instructions per Cycle)
- Branch miss ratio

Recording pipeline-level stall detail

```
perf record -e cycles:u -e instructions:u -e branches:u -e branch-misses:u ./a.out
perf report
```

To examine micro-op dispatch and memory subsystem:

```
perf stat -e \
  cycles, \
  instructions, \
  L1-dcache-load-misses, \
  LLC-load-misses, \
  branch-misses, \
  idq_uops_not_delivered.core, \
  uops_issued.any, \
  resource_stalls.any \
  ./a.out
```

Key Interpretation:

- **IPC < 1** typically indicates stalls or poor parallelism.
- **LLC-load-misses** track working-set spill outside L2.
- **idq_uops_not_delivered** identifies front-end starvation.
- **resource_stalls.any** tracks backend pipeline congestion.

C.5 Annotated Disassembly for Pipeline Attribution

To correlate microarchitectural events with specific instructions:

```
perf record -g ./a.out
perf annotate
```

This overlays event frequency on machine instructions.

- Vectorization success corresponds to sustained **wide-lane instruction clusters**.
- Register allocation pressure reveals itself as **spill/reload traffic**.
- Branch misprediction appears as **high-frequency penalties** surrounding conditional jumps.

C.6 Integrated Diagnostic Workflow

Step	Tool	Objective
1	`readelf`	Validate symbols, relocations, dynamic loader linkage
2	`objdump`	Inspect final emitted instruction sequence

Step	Tool	Objective
3	`gdb`	Reconstruct control flow and call boundaries
4	`perf stat`	Collect macro performance metrics
5	`perf record/annotate`	Attribute stalls and latency sources to exact instructions
6	Compare with SSA/GIMPLE dumps	Determine whether performance loss originates from algorithmic structure, missed optimization opportunities, or architecture-level constraints

The result is a **closed-loop analysis** linking:

- Compiler-level decision trace
- Binary-level execution structure
- Hardware-level pipeline behavior

Purpose Restated

This appendix provides a reproducible workflow to:

- Confirm compiler-intended transformations are present and correct,
- Attribute runtime behavior to precise machine instructions,
- Validate symbol linkage, calling convention compliance, and ABI stability,
- Identify microarchitectural bottlenecks and optimization opportunities.

The integration of symbolic reasoning and instruction-level performance analysis is a prerequisite for **trusted high-performance C++ systems development**.

Appendix D - Linker Scripts and ELF Structural Control

This appendix presents the mechanisms required to explicitly control ELF binary layout when targeting Linux x86-64 with GCC and the GNU linker (`ld`). By default, GCC relies on built-in linker scripts that define section placement and address assignment. System-level software, kernel components, static runtimes, and embedded deployments frequently require **deterministic and analyzable** layout, making manual control essential.

The material here describes the structure of a custom linker script, the semantics of ELF sections, the impact of dynamic linking relocations, and constraints related to position-independent execution. The treatment assumes familiarity with ELF parsing tools (`readelf`, `objdump`) and the ABI rules described in Appendix A.

D.1 Structure of a Minimal Linker Script

The GNU linker script grammar is declarative: memory regions and output sections are defined, and input object sections are mapped accordingly.

Example (annotated):

```
/* Define ELF entry point */
ENTRY(_start)

/* Default memory region for userland process */
SECTIONS
{
    /* Code segment */
    .text : ALIGN(16) {
        KEEP(*(.init))            /* CRT initialization entry */
        *(.text .text.* .gnu.linkonce.t.*)
```

```
    }

    /* Read-only constants */
    .rodata : ALIGN(16) {
        *(.rodata .rodata.* .gnu.linkonce.r.*)
    }

    /* Global/static data (initialized) */
    .data : ALIGN(16) {
        *(.data .data.* .gnu.linkonce.d.*)
    }

    /* Global/static data (zero-initialized) */
    .bss : ALIGN(16) {
        *(COMMON)
        *(.bss .bss.* .gnu.linkonce.b.*)
    }

    /* C++ static constructors / destructors */
    .init_array : ALIGN(8) {
        PROVIDE(__init_array_start = .);
        KEEP(*(.init_array .init_array.*))
        PROVIDE(__init_array_end = .);
    }

    .fini_array : ALIGN(8) {
        PROVIDE(__fini_array_start = .);
        KEEP(*(.fini_array .fini_array.*))
        PROVIDE(__fini_array_end = .);
    }

    /* Thread-local storage */
    .tdata : ALIGN(8) { *(.tdata .tdata.*) }
```

```
    .tbss  : ALIGN(8) { *(.tbss .tbss.*) }
}
```

Key principles:

- Section ordering directly influences page locality and execution bandwidth.
- `KEEP()` prevents startup metadata from being removed by garbage-collection (`--gc-sections`).
- `PROVIDE()` emits exported symbols enabling C++ runtime initialization sequences.

D.2 Relocation Sections and Dynamic Linking Semantics

Dynamic binaries contain relocation records that resolve symbol addresses at load time. Relevant ELF sections:

Section	Purpose
`.rela.dyn`	Global relocations for data and GOT
`.rela.plt`	Relocations used by PLT trampolines
`.got` / `.got.plt`	Indirection tables used by loaders
`.plt`	Late-binding trampoline stubs

Behavior:

- If `-fPIC` or `-pie` is enabled, function and global references are emitted using **PC-relative** addressing and indirect resolution via GOT/PLT.
- For **static binaries**, relocation fixups occur at link time, and `.plt` and `.got.plt` are omitted.

Inspect relocations:

```
readelf -r program
readelf -d program | grep RELA
```

This is essential when verifying:

- Symbol interposition rules,
- IFUNC dispatch,
- Position-independent address arithmetic.

D.3 Segment Alignment and Protection Control

ELF segments are mapped page-wise. Section-to-segment mapping controls permissions: Typical layout (simplified):

Segment	Permissions	Contains
PT_LOAD(text)	RX	.text, .rodata
PT_LOAD(data)	RW	.data, .bss, TLS regions
PT_TLS	RW	.tdata, .tbss

Control using:

```
PHDRS
{
    text PT_LOAD FLAGS(R X);
    data PT_LOAD FLAGS(R W);
    tls PT_TLS FLAGS(W);
}
```

```
SECTIONS {
    .text : { ... } :text
    .rodata : { ... } :text
    .data : { ... } :data
    .bss : { ... } :data
    .tdata : { ... } :tls
    .tbss : { ... } :tls
}
```

This allows enforcing:

- W^X memory policy,
- Executable separation for audit/security,
- Cache locality tuning for performance-sensitive systems.

D.4 Hot and Cold Code Separation

Modern GCC marks branch-predicted unlikely blocks:

- `.text.unlikely` for cold paths,
- `.text.hot` for frequently executed sequences.

Enable code section splitting:

```
g++ -O3 -freorder-blocks-and-partition
```

Linker script mapping:

```
.text.hot : ALIGN(16) { *(.text.hot .text.hot.*) }
.text.unlikely : ALIGN(16) { *(.text.unlikely .text.unlikely.*) }
```

Benefits:

- Improved I-cache residency for hot loops,
- Cold paths moved outside primary fetch footprint,
- Reduced branch target collision for high-frequency loops.

D.5 Position-Independent Code and Addressing Rules

For shared libraries (`-fPIC`):

- Use **RIP-relative** addressing for globals.
- Access to data objects occurs via GOT indirection.

For PIE executables (`-pie`):

- The entire binary is relocatable at runtime.
- Startup loader assigns a randomized base address (ASLR).

Exceptions:

- Bare-metal, kernel, or VMM code may use absolute addressing.
- Must disable relocations and PIE:

```
g++ -nostdlib -static -fno-pic -no-pie -Wl,-T,linker.ld
```

D.6 Verification Workflow

1. Inspect section and segment layout:

```
readelf -lW a.out
readelf -SW a.out
```

2. Confirm GOT/PLT presence or elimination:

```
objdump -d a.out | grep plt
```

3. Verify constructor and TLS boundaries:

```
nm -n a.out | grep init_array
```

4. Validate page protections:

```
/proc/<pid>/maps
```

Objective Restated

This appendix equips the reader to:

- Construct **deterministic memory layouts**,
- Control **runtime initialization semantics**,
- Enforce **security and performance-aware mapping policies**,
- Produce **statically analyzable and tightly constrained binaries** for high-assurance environments.

These capabilities are foundational in embedded runtimes, kernel-level components, language runtimes, and high-performance distributed execution systems.

Appendix E - Bare-Metal C++ Runtime Templates

Bare-metal execution requires constructing a minimal runtime that performs tasks normally handled by the C library, dynamic loader, and kernel. This appendix provides reference templates illustrating how to establish an ABI-conformant execution environment, initialize memory regions, invoke static constructors, provide memory allocation facilities, and ensure deterministic termination. These templates allow C++ to execute on hardware or emulated platforms without glibc, without `crt1.o`, and without dynamic loader support.
The routines here assume:

- System V AMD64 ABI semantics (register assignments, stack alignment),
- A flat virtual/physical address model (no paging assumptions),
- No kernel services unless explicitly provided.

E.1 `_start` Entry and Stack Establishment

At process or boot entry, no stack frame is present and no runtime state is initialized. The entry routine must:

1. Set up a valid stack,
2. Zero-initialize `.bss`,
3. Call global constructors via `.init_array`,
4. Transfer control to `main`,
5. Execute destructors before termination, if required.

Example — handwritten `_start` in assembly:

```
    .global _start
_start:
    /* Assume bootloader or environment sets RSP to usable stack top */

    /* Zero .bss */
    lea _bss_start(%rip), %rdi
    lea _bss_end(%rip),   %rsi
    call memset_range_zero

    /* Run global constructors */
    lea __init_array_start(%rip), %rdi
    lea __init_array_end(%rip),   %rsi
    call run_constructors

    /* Call main() */
    call main

    /* Call destructors */
    lea __fini_array_start(%rip), %rdi
    lea __fini_array_end(%rip),   %rsi
    call run_destructors

    /* Program exit - no OS assumes shutdown */
1:  jmp 1b
```

This entry sequence is conformant with C++ initialization rules while remaining independent of external runtimes.

E.2 `.bss` Clearing and Constructor Invocation

Zero-initialization ensures that all storage-duration objects in `.bss` start in the correct state.

Minimal zeroing implementation:

```
extern "C" void memset_range_zero(void* begin, void* end) {
    unsigned char* p = static_cast<unsigned char*>(begin);
    while (p < end) {
        *p++ = 0;
    }
}
```

Global constructor execution:

```
extern "C" void run_constructors(void** begin, void** end) {
    for (void** fn = begin; fn < end; ++fn) {
        reinterpret_cast<void(*)()>(*fn)();
    }
}

extern "C" void run_destructors(void** begin, void** end) {
    for (void** fn = end; fn > begin; ) {
        (--fn, reinterpret_cast<void(*)()>(*fn)());
    }
}
```

This matches the initialization semantics described in the Itanium C++ ABI.

E.3 Minimal Heap for Dynamic Allocation

A simple bump allocator provides deterministic and threadless memory allocation. This is sufficient for embedded execution without fragmentation control.

```
static unsigned char* heap_base;
static unsigned char* heap_limit;
static unsigned char* heap_ptr;

extern "C" void heap_init(void* base, std::size_t size) {
    heap_base = static_cast<unsigned char*>(base);
    heap_limit = heap_base + size;
    heap_ptr = heap_base;
}

extern "C" void* malloc(std::size_t n) {
    unsigned char* r = heap_ptr;
    if (r + n > heap_limit) return nullptr;
    heap_ptr += n;
    return r;
}

extern "C" void free(void*) { /* no-op */ }
```

To enable `new`/`delete`:

```
void* operator new(std::size_t n) { return malloc(n); }
void operator delete(void*) noexcept {}
```

This allocator is stable under single-threaded deterministic workloads.

E.4 System-Independent Console Output (UART / MMIO)

Bare-metal output typically targets a memory-mapped peripheral interface.
Example — generic byte-write to MMIO UART:

```
static volatile unsigned char* const UART0 = reinterpret_cast<unsigned
↪  char*>(0x10000000);
```

```
extern "C" void putc(char c) {
    *UART0 = static_cast<unsigned char>(c);
}

extern "C" void puts(const char* s) {
    while (*s) putc(*s++);
}
```

This function set forms the basis for diagnostics, logging, and test output.

E.5 Program Termination

If no operating environment is present, program termination must resolve to a controlled halt.

```
extern "C" void abort() {
    for(;;) { __asm__ volatile("hlt"); }
}
```

If running under QEMU, halting or writing to a debug port may be appropriate.

E.6 Example: Minimal Bare-Metal C++ Executable Build

Compile without CRT and without glibc:

```
g++ -nostdlib -ffreestanding -fno-exceptions -fno-rtti \
    -Wl,-T,linker.ld startup.o runtime.o main.cpp -o baremetal.elf
```

The linker script must define:

```
_bss_start
_bss_end
```

```
__init_array_start
__init_array_end
__fini_array_start
__fini_array_end
```

as described in Appendix D.

Outcome

This appendix provides the **foundation required to execute C++ on bare hardware**, without external libraries, dynamic loaders, or kernel support. By defining:

- Entry sequencing,
- Memory initialization,
- Constructor/destructor coordination,
- Minimal heap provisioning,
- Low-level I/O interfaces,

the developer gains full control of execution semantics, memory topology, and binary layout.
This runtime layer forms the stepping stone for:

- Embedded firmware,
- Microkernel and hypervisor construction,
- OS development research,
- Real-time and deterministic control environments.

The principles scale directly to both single-core and multi-core architectures.

Appendix F - Performance and Microarchitectural Reference

This appendix provides a compact, engineering-oriented reference for interpreting disassembly against contemporary x86-64 cores. Values are representative of **Skylake-class (SKL/SKL-X/CL)**, **Ice Lake client/server (ICL/ICX)**, and **Zen-class (Zen2/Zen3/Zen4)** cores. Where silicon/stepping and frequency scaling materially affect numbers, ranges or normalized costs (cycles per µop / per cache line) are given. Use these tables to reason about throughput ceilings, latency bottlenecks, and the side-effects of vector width and memory hierarchy.

F.1 Execution Ports and Issue Structure (High-Level)

F.1.1 Intel Skylake-class (SKL/CL)

- **Front-end**: up to **µops/cycle** decode; µop cache delivers up to **µops/cycle**.
- **Back-end ports** (common view):
  - **p0/p1**: vector/FP ALU (FMA, add, mul)
  - **p5**: integer ALU (also branch)
  - **p2/p3**: load address gen (AGU) and load data (2 loads/cycle total)
  - **p4**: store data
  - **p7**: store address gen (AGU)
- **Loads/stores per cycle**: typically **loads + 1 store** (1 store data + 1 store address).

- **Branch**: one per cycle (p5), with fused-domain front-end for simple cmp+jcc pairs.

F.1.2 Intel Ice Lake (ICL/ICX)

- **Front-end**: wider rename/issue, improved µop cache efficiency.
- **Back-end**:
  - **AGUs**: **load + 1 store** address in the same cycle more reliably than SKL.
  - Additional improvements to vector permutes and gather/scatter throughput.
- **Vector**: AVX-512 supported on many SKUs; per-core FMA width increases with 512-bit units.

F.1.3 AMD Zen-class (Zen2/Zen3/Zen4)

- **Front-end**: µop cache (Zen2+), 4-wide decode typical.
- **Back-end**:
  - **load + 1 store** per cycle sustained (Zen2/3), AGU availability depends on generation.
  - FP pipelines: **2× FMA** (Zen2/3) with balanced add/mul throughput.
  - Zen4 improves permute/broadcast and vector integer throughput.

Implications: peak loop body throughput is often bounded by **AGU availability (address generation)**, **memory ports** (load/store), or **vector FMA/add units**, not by scalar ALUs.

F.2 Latency and Reciprocal Throughput (Representative)

Values below are typical at nominal clocks. Always validate on target hardware.

Operation (scalar unless noted)	**Skylake**	**Ice Lake**	**Zen3**
Integer add `add r64,r64`	1c / 0.25c	1 / 0.25	1 / 0.25
Integer mul `imul r64`	3 / 1	3 / 1	3 / 1
FP add `vaddpd ymm`	4 / 0.5	3–4 / 0.5	3–4 / 0.5
FP mul `vmulpd ymm`	4 / 0.5	3–4 / 0.5	3–4 / 0.5
FMA `vfmadd231pd ymm`	4–5 / 0.5	4 / 0.5	4 / 0.5
Load (L1 hit)	4–5 latency	~4	4–5
Store (to store buffer)	~1 retire	~1	~1
Shuffle `vpermilpd ymm`	3–6 / 1	3–4 / 0.5–1	3–5 / 0.5–1
Gather `vpgatherdd ymm`	tens of cycles; mem-lat bound	improved but mem-bound	improved but mem-bound

`c = cycles;` "/" separates **latency / reciprocal throughput**. For wide vectors, latency often stays similar; throughput scales with width until limited by ports, rename, or power/freq constraints.

F.3 Cache & Memory Hierarchy (Rules of Thumb)

F.3.1 Sizes, associativity, and typical access

Level	Capacity (typical)	Assoc	Line	Latency (cycles)
L1D	32 KiB/core	8-way	64 B	~4
L2	256–1280 KiB/core	4–12	64 B	~10–14 (ICL tends lower)
LLC	2–64 MiB / shared	16–24	64 B	~35–80
DRAM	tens of GiB/s	—	—	~120–300 (NUMA-, freq-, page-policy-dependent)

F.3.2 Bandwidth ceilings (steady-state, single core, streaming)

- **L1D**: approach **2×64 B loads + 1×64 B store per N cycles**; practical ~50–90+ GB/s depending on vector width and core.
- **L2**: ~20–40 GB/s.
- **DRAM** (single core): **10–30 GB/s** depending on memory channels, prefetch, and stride.

Rule: If a loop consumes **>1 cache line per ~3–5 cycles**, you are quickly **bandwidth-bound**, regardless of arithmetic throughput.

F.4 TLB and Page Walk Costs

TLB	Coverage (typical)	Miss Cost
L1 DTLB	~64 entries (4 KiB pages)	Falls to L2 TLB; tens of cycles
L2/STLB	~1–2k entries	Page walk if miss

TLB	Coverage (typical)	Miss Cost
Page walk	4-level (4 KiB pages)	~100–200 cycles; parallelism limited
Huge pages	2 MiB / 1 GiB	Fewer levels → reduced miss rate and walk cost

Guidance: Regularize strides, prefer **contiguous** access, and consider **MiB huge pages** for large arrays with streaming access to reduce TLB pressure.

F.5 AVX2 / AVX-512 Alignment & Access Constraints

- **Alignment**: 32-byte alignment for AVX2 (`ymm`), **64-byte** preferred for AVX-512 (`zmm`) to minimize split-line penalties and enable aligned loads/stores.
- **Misalignment penalties**:
  - Crossing a 64-B line adds an extra load; repeated line splits degrade sustained throughput.
  - Gather/scatter are **latency dominated** by memory; use only when structure prevents SoA refactor.
- **Non-temporal stores** (`vmovntps/pd`): beneficial for write-only streams that outstrip cache reuse; avoid polluting caches and can increase sustained DRAM bandwidth.

F.6 Mixed-Width Transitions and Frequency Behavior

- **SSE AVX mixing**: On older Intel parts (pre-ICL), mixing SSE and AVX frequently can trigger domain transitions; keep a kernel homogeneously

vectorized.

- **AVX-512 down-clock**: Many Intel SKUs reduce core frequency when executing heavy AVX-512 to stay within power/thermal envelopes. Throughput may still improve for math-dense kernels; measure.
- **Mitigation**:
  - Isolate wide-vector kernels in time (batch), or constrain vector width (`-mno-avx512f` or use AVX2) if frequency loss dominates.
  - Keep **hot loops** free of scalar instructions when vector lanes are active (avoid reformatting between scalar/vector inside the loop body).

F.7 Roofline and "Cycles per Cache Line" Heuristics

F.7.1 Operational intensity

$$I = \frac{\text{FLOPs}}{\text{Bytes moved}}$$

Compare **I** against attainable bandwidth to determine if the kernel is **compute-bound** or **memory-bound**.

F.7.2 Cycles per CL model

For a streaming loop touching **N** cache lines per iteration with sustained **B** bytes/cycle from the relevant level:

$$\text{cycles/iter} \approx \frac{64N}{B}$$

If this exceeds arithmetic throughput time, the kernel is bandwidth-bound.

F.8 Practical Diagnostics: What to Check in Annotated Disassembly

1. **Load/store grouping**: aim for **loads + 1 store** per cycle sustained.
2. **AGU saturation**: address patterns that exceed available AGUs throttle throughput.
3. **Vector homogeneity**: avoid scalar ops within vectorized loop bodies.
4. **Shuffle density**: excessive permutes often dominate port usage; refactor data layout (SoA) to reduce permutes/gathers.
5. **Prefetch**: regular, unit-stride accesses let hardware prefetch hide most of L2 latency; irregular strides may need software prefetch.

F.9 Quick Reference Tables

F.9.1 Vector width & elements per 64-B line

Type	AVX2 (256b)	AVX-512 (512b)	Elements/CL
`float` (4 B)	8	16	16
`double` (8 B)	4	8	8
`int32_t`	8	16	16
`int64_t`	4	8	8

F.9.2 Sustained arithmetic ceilings (per core, idealized)

Kernel	SKL	ICL	Zen3
DP FMA (AVX2)	up to **16 flops/cycle** (2 FMAs × 4 lanes)	similar or better	similar
DP FMA (AVX-512)	up to **32 flops/cycle** (2 FMAs × 8 lanes)	supported	N/A (no AVX-512 on Zen2/3)

Actuals depend on port conflicts, loads/stores, and frequency behavior.

F.10 Engineering Guidance (Checklist)

- **Data layout first**: SoA for vector kernels; 64-B alignment on hot arrays.
- **One width per kernel**: choose SSE/AVX2/AVX-512 and keep loops uniform.
- **Balance memory ops**: target **2L/1S** per cycle; minimize line splits.
- **Control working set**: fit inner loops into **L1**, outer tiles into **L2**.
- **Measure**: verify with `perf stat` (IPC, L1/LLC misses), `perf annotate` for per-instruction hotspots.
- **Consider pages**: huge pages for large, streaming datasets to reduce TLB misses.

Purpose Restated

This appendix supplies **numbers and rules** to translate a hot loop's disassembly into expected throughput and latency, recognize when execution is **port/AGU-bound** vs **memory/TLB-bound**, and make data-driven choices about vector width, layout, and tiling. Use it as your on-bench reference while iterating on high-performance C++ kernels compiled with GCC on modern x86-64.

Appendix G - Verified Object Model Layouts

This appendix records canonical C++ object layouts as emitted by GCC (G++ 10+) for the Itanium C++ ABI on Linux x86-64. The intent is verification: correlate source declarations with binary shape (object memory, vptr locations, VTable organization, RTTI objects, and adjustment thunks) to validate ABI compliance, support reverse analysis, and drive advanced debugging.
Assumptions:

- System V AMD64 ABI (Appendix A) for data layout and calling convention.
- Itanium C++ ABI for object model, mangling, RTTI, and exception machinery.
- GCC defaults: new/delete from libstdc++, EH enabled, RTTI on.

G.1 Standard Layout, Trivial Types, and POD Aggregates

For standard layout and trivial aggregates, GCC lays out members with natural alignment and no interstitial control fields.

```
struct S {
  int    a;     // 4B
  double b;     // 8B
  char   c;     // 1B
};              // sizeof(S) == 24 on x86-64
```

Memory (little-endian, byte offsets):

```
0x00: a (4) | padding (4)
0x08: b (8)
0x10: c (1) | padding (7)
```

Rules:

- Aggregates have no hidden headers or vptrs.
- Alignment equals max alignment of members (`alignof(S) == 8` here).
- Empty base optimization (EBO) applies when a class with no data (other than its type identity) is used as a base; its size may collapse to 1 or be subsumed at offset 0 in a derived object when permitted by the ABI.

Verification:

- `static_assert(std::is_standard_layout_v<S> && std::is_trivial_v<S>);`
- `offsetof(S, a)==0`, `offsetof(S, b)==8`, `offsetof(S, c)==16`.

G.2 Single Inheritance (Non-Virtual) with Polymorphism

Polymorphic classes contain a single **vptr** (pointer to VTable) at object offset **0**.

```
struct Base {
  virtual ~Base();
  virtual int f() const;
  int x;                        // data follows vptr
};

struct Derived : Base {
  int y;
  int f() const override;
};
```

Object layout (x86-64):

```
Base:
  +0x00: vptr (8) -> &VTable_Base[0]
  +0x08: x (4)
```

```
  +0x0C: padding (4)
  sizeof(Base) == 16

Derived:
  +0x00: vptr (8) -> &VTable_Derived[0]
  +0x08: x (4)
  +0x0C: padding (4)
  +0x10: y (4)
  +0x14: padding (4)
  sizeof(Derived) == 24
```

VTable shape (primary vtable for the most derived object):

```
VTable_Derived:
  [0]: ptr-to-typeinfo (&typeinfo for Derived)
  [1]: ptrdiff_t offset-to-top (0 for primary)
  [2]: &Derived::~Derived (deleting destructor)
  [3]: &Derived::~Derived (complete destructor)
  [4]: &Derived::f
  ...
```

Notes:

- The first two slots form the VTable header (RTTI pointer, offset-to-top).
- Destructors appear before other virtuals per Itanium order.
- `offset-to-top == 0` for primary vptr in a complete object.

G.3 Multiple Inheritance (Non-Virtual Bases)

With multiple non-virtual bases, the subobject of the first listed base occupies the object prefix; subsequent bases follow with their own data but no additional vptrs unless polymorphic subobjects are distinct.

```
struct A { virtual ~A(); int a; };
struct B { virtual void g(); int b; };
struct C : A, B { int c; };
```

Layout:

```
C (complete object):
  [A-subobject]
    +0x00: vptr_A' -> primary VTable_C (A view)
    +0x08: a
  [B-subobject]
    +0x10: vptr_B' -> secondary VTable_C (B view)
    +0x18: b
  [C-own]
    +0x20: c
  sizeof(C) == 0x28 (alignment/padding may vary)
```

Key properties:

- **Two vptrs** exist: one at offset 0 (A view), another at the start of the B-subobject.
- The **secondary VTable** (B view) has a nonzero **offset-to-top** ($-0x10$ here), enabling `dynamic_cast` from `B*` to `C*` through the negative adjustment.
- Virtual function pointers in each vtable are adjusted thunks as required to rebase `this` when calls originate from the corresponding base view.

G.4 Virtual Inheritance

Virtual bases are stored once, at an implementation-chosen offset placed after the non-virtual part of the most-derived object. Access uses **virtual base offsets** stored in the VTable.

```
struct V { virtual ~V(); int v; };
struct A : virtual V { int a; };
struct B : virtual V { int b; };
struct C : A, B { int c; };
```

Schematic layout:

```
C (complete object):
  [Primary subobject] (A is primary if first listed)
    +0x00: vptr_C(A-view)
    +0x08: A::a
  [B-subobject]
    +0x10: vptr_C(B-view)
    +0x18: B::b
  [C-own]
    +0x20: C::c
  [Virtual base V]   <-- placed once
    +0x28: vptr_C(V-view)
    +0x30: V::v
```

VTable (primary view) contains:

- RTTI pointer
- offset-to-top (0)
- virtual function slots
- **Virtual base offset entries** (vcall/vbase offsets) enabling dynamic adjustment to reach `V` from any subobject view at runtime, independent of the physical placement chosen by the linker.

Consequences:

- Any pointer to a base subobject carries an implicit view; calls may require **this-adjusting thunks**.
- `dynamic_cast` consults RTTI graph and vbase offsets to compute unique addresses or report failure.

G.5 VTable, Thunks, and This-Adjustment

For non-leftmost bases or virtual bases, GCC may emit thunks that add a constant delta to `this` before tail-calling the real implementation:

```
; thunk for B::g() in C when called via B* (delta = -0x10)
C::_ZThn16_N1B1gEv:
  lea rdi, [rdi - 0x10]     ; adjust this to C*
  jmp C::g()                ; tail call keeps EH tables intact
```

- Thunks are listed in the vtable entries corresponding to that base view.
- Thunks preserve EH personality and CFI expectations; they are leaf glue.

G.6 RTTI Objects (`typeinfo`) and Relationships

Each polymorphic type has a `typeinfo` object:

```
VTable header slot 0  -->  &typeinfo for most-derived type
```

`typeinfo` encodes:

- Mangled name of the type,
- Base class list and virtual base graph (via ABI-defined structures),
- Used by `typeid` and `dynamic_cast`.

Offsets:

- `offset-to-top` (slot 1) is a signed `ptrdiff_t` applied to a subobject pointer to reach the most-derived object.
- Secondary vtables carry their own offset-to-top values.

G.7 Composite Reconstruction from Raw Memory

Given only a data pointer and vptr, reconstruct the object view:

1. Read vptr at `[p]` → `vptr0`.
2. Resolve `vptr0[-1]` (slot −1) → `offset-to-top`.
3. Compute `top = p + offset-to-top`.
4. Inspect `vptr0[-2]` (slot −2) → RTTI pointer; identify the dynamic type.
5. Use ABI rules or known class definitions to parse subobject boundaries.

For multiple/virtual inheritance:

- Secondary vptrs appear at known subobject offsets; each carries its own header.
- Virtual base offsets are obtained from the vtable's vbase table; add to `top` to locate virtual bases.

G.8 Validation Patterns (GDB / objdump)

Recommended checks:

- Confirm vptr at object offset 0 for primary base; locate secondary vptrs at non-zero offsets.

- Dump vtables (`objdump -s -j .rodata`) and identify:
  - `&typeinfo`,
  - `offset-to-top`,
  - sequence of function pointers and thunks.
- In GDB:
  - `(gdb) p/x *(void**)obj` → vptr
  - `(gdb) p/x ((void**)vptr)[-1]` → offset-to-top
  - `(gdb) p ((std::type_info*) ((void**)vptr)[-2])->name()` (symbolization available in non-stripped builds)

G.9 Edge Cases and Notable Details

- **EBO (Empty Base Optimization)**: Empty non-virtual bases may occupy no additional space in the most-derived object's layout.
- **Final classes / devirtualization**: Optimization does not remove vptrs for externally visible types; ABI requires stable layout even if calls devirtualize internally.
- **Multiple identical base subobjects**: Disambiguation via qualified casts; RTTI encodes the base graph for unambiguous `dynamic_cast`.
- **Diamond with virtual base**: Only one `V` instance exists in the most-derived; base subobjects carry vbase offsets to reach it.

G.10 Minimal Worked Examples

G.10.1 Two-base non-virtual

```
struct A { virtual ~A(); int a; };
struct B { virtual ~B(); int b; };
struct C : A, B { int c; };
```

Expected:

- vptr at offset 0 (A view), secondary vptr at offset `sizeof(A)` (B view).
- Two vtables for C: primary (A view), secondary (B view).
- `offset-to-top` in secondary vtable = `-ptrdiff_t(sizeof(A))`.

G.10.2 Virtual diamond

```
struct V { virtual ~V(); int v; };
struct A : virtual V { int a; };
struct B : virtual V { int b; };
struct C : A, B { int c; };
```

Expected:

- Primary vptr at offset 0 (A view) with vbase table entries to locate `V`.
- C contains one `V` instance; both A and B access it via vbase offsets.
- Thunks adjust `this` from A/B views to C where needed.

G.11 Practical Checklist

- **Primary vptr at 0** for polymorphic complete objects.
- **Secondary vptrs** start each non-leftmost base subobject.
- **offset-to-top** negative for secondary views; zero for primary.
- **RTTI slot** always immediately preceding offset-to-top in the vtable.
- **Thunks** appear where cross-view `this` adjustment is needed; tail-call preserves CFI.
- **Virtual bases** resolved via vbase offsets from the vtable of the current view.

Objective Restated

This appendix provides verified patterns for GCC's Itanium C++ ABI object layouts. Use them to:

- Validate ABI stability across releases,
- Reconstruct composite objects from memory alone,
- Interpret vtable headers and thunks during reverse engineering,
- Guarantee correctness when interfacing hand-written assembly, JIT stubs, or binary patching with polymorphic C++ objects.

Appendix H - Full Compilation and Optimization Case Study

This appendix presents a reproducible, end-to-end walk from C++ source to an optimized ELF binary, correlating each compiler stage with the emitted machine code and measured runtime behavior. The goal is to verify that semantic intent is preserved, optimizations are justified by a cost model, and ABI contracts remain intact.

The example is deliberately simple but non-trivial: a reduction kernel with a branchless transform and a hot inlinable helper. Build on Linux x86-64 with GCC 10.

H.1 Source Program

```
// file: case.cpp
#include <cstddef>
#include <cstdint>
#include <cmath>

static inline double transform(double x) noexcept {
    // small nonlinearity to provoke vector math + FMA opportunities
    return std::sqrt(x * x + 1.0);
}

extern "C"
double reduce_sum(const double* __restrict a, std::size_t n) {
    double acc = 0.0;
    for (std::size_t i = 0; i < n; ++i) {
        acc += transform(a[i]) * 0.5;
    }
    return acc;
}
```

Build variants (instrumented):

```
# Baseline optimized + dumps
g++ -O3 -march=native -fno-exceptions -fno-rtti \
    -fdump-tree-original-raw -fdump-tree-optimized -fdump-tree-ssa \
    -fdump-tree-vect -fdump-rtl-expand -fdump-rtl-final \
    -S -o case.s -c case.cpp

# LTO + PGO path (optional)
g++ -O3 -flto -fprofile-generate -march=native -c case.cpp -o case.gen.o
g++ -O3 -flto -fprofile-generate -o app.gen case.gen.o
./app.gen  # run with representative data to emit .gcda
g++ -O3 -flto -fprofile-use -march=native -o app.opt case.cpp
```

H.2 Stage 1 — AST and Semantic Graph (`-fdump-tree-original-raw`)

Artifacts: `case.cpp.003t.original`.
What to verify:

- Function prototypes: `extern "C" double reduce_sum(const double*, size_t)`.
- `transform` is `static inline` and `noexcept`: eligible for inlining at O3.
- Loop canonicalization: induction `i` from 0 to `n`, strict forward progress.

Actionable checks:

- Confirm no implicit temporaries that would block vectorization (e.g., by-value aggregates).
- Confirm `__restrict` propagated on pointer `a`.

H.3 Stage 2 — GIMPLE + SSA + CFG

Artifacts: `case.cpp.optimized`, `case.cpp.ssa`, `case.cpp.vect`.
Key markers:

- **Inlining**: `transform` should be eliminated as a separate call; look for its body folded into the loop body in `optimized`.
- **SSA**: `acc` becomes -merged at the loop header; `i` is an induction variable with a single definition chain.
- **Alias and TBAA**: `__restrict` on `a` reduces alias pessimism for the load.
- **Vectorization report** (`-fdump-tree-vect`): expect something like "loop vectorized width=4/8" depending on ISA.

What to read:

- In `vect` dump: lanes, data-ref analysis (stride = 8 bytes), cost model acceptance, epilogue handling for `n % VL`.

H.4 Stage 3 — Midend Optimization Decisions

Transformations to confirm in dumps:

- **Constant folding**: multiply by `0.5` may fuse into FMA or be represented as `* 0x3fe0000000000000`.
- **Strength reduction**: address computation uses scaled index addressing.
- **Loop distribution/fusion**: not applicable here; confirm preserved single hot loop.

- **Math-library lowering**: `sqrt` for vectors may map to hardware `vsqrtpd` or a vector helper depending on target and flags.

Decision visibility:

- `-fopt-info-vec` should explicitly state whether `sqrt` vectorization is applied via hardware instruction or internal vector libcall.

H.5 Stage 4 — RTL Emission (Pre/Post RA)

Artifacts: `case.cpp.expand`, `case.cpp.final`.
What to confirm in `expand`:

- Loads from `a[i]` as `MEM[base + index*8]` with vector width if vectorized.
- Temporary virtual regs for vector accumulators.
- Canonical loop control with compare + branch based on vector iteration count and a scalar epilogue.

What to confirm in `final`:

- Register allocation stability: vector regs bound to `ymm`/`zmm` (target-dependent), minimal spills.
- Addressing modes legalized (no out-of-range displacements).
- Peephole/sched patterns: fused compare+branch; hoisted invariants (scale, constants).

H.6 Stage 5 — Final Assembly and Relocation Dump

Generate annotated assembly and relocations:

```
objdump -dr -Mintel app.opt | less
readelf -rW app.opt
```

Disassembly expectations (AVX2 example):

```
.Lloop:
    vmovupd   ymm1, YMMWORD PTR [rdi+rax*8]   ; load 4 doubles
    vmulpd    ymm2, ymm1, ymm1                ; x*x
    vaddpd    ymm2, ymm2, YMMWORD PTR [.LC1]  ; +1.0
    vsqrtpd   ymm2, ymm2                      ; sqrt
    vmulpd    ymm2, ymm2, YMMWORD PTR [.LC05] ; *0.5
    vaddpd    ymm0, ymm0, ymm2                ; accumulate
    add       rax, 4
    cmp       rax, rsi
    jb        .Lloop
```

Relocations:

- For a non-PIC executable, expect minimal dynamic relocations; constants likely in `.rodata` without GOT indirection.
- For PIE/`-fPIC`, loads of constants via RIP-relative addressing/GOT as appropriate.

H.7 Stage 6 — Runtime Performance Trace (perf)

Test driver (generate input, call `reduce_sum`):

```
#include <vector>
#include <random>
extern "C" double reduce_sum(const double*, std::size_t);

int main() {
    std::vector<double> a(20'000'000);
    std::mt19937_64 rng(0);
    std::uniform_real_distribution<double> U(0.0, 10.0);
    for (auto& x : a) x = U(rng);
    volatile double s = reduce_sum(a.data(), a.size());
    (void)s;
    return 0;
}
```

Measure:

```
perf stat -d ./app.opt
perf record -g ./app.opt
perf annotate
```

Readouts to correlate:

- **IPC** near 1.2–2.0 for a balanced vector kernel on a modern core.
- **L1/L2 miss** rates low for streaming access with unit stride.
- **idq_uops_not_delivered** small if front-end not starved.
- **uops_issued.any** proportional to vector body; check that spill traffic is negligible.
- Annotate: verify hot basic block aligns with vector loop; instruction with highest sample share should be `vmovupd`/`vsqrtpd`/`vaddpd` in a balanced ratio.

If AVX-512 is available and enabled, note possible frequency changes; still expect higher per-iteration work.

H.8 Stage 7 — ABI Verification (`readelf -Ws`)

```
readelf -Ws app.opt | grep -E 'reduce_sum|transform'
```

Expected:

- `reduce_sum` exported as a global **default** symbol with unmangled C name (due to `extern "C"`).
- No visible `transform` symbol if fully inlined; if present (e.g., under different build), its visibility should be local/hidden or eliminated.

Call signature (System V AMD64):

- `const double*` in **RDI**, `size_t` in **RSI**.
- Return in **XMM0**.

Check `readelf -lW` for segment permissions: `.text` RX, `.rodata` R, `.data`/`.bss` RW; confirm PIE vs non-PIE as intended.

H.9 Cross-Stage Consistency Matrix

Question	Where to verify	Expected signal
Was `transform` inlined?	`optimized` dump; symbol table	Body merged into loop; symbol removed or local
Was the loop vectorized?	`vect` dump, assembly	Vector width lanes; `vaddpd`/`vsqrtpd`

Question	Where to verify	Expected signal
Any missed optimization due to aliasing?	`ssa` (TBAA notes), `perf`	No extra reloads; high IPC
Register pressure acceptable?	`rtl.final`, `perf annotate`	No spills in hot block
ABI stable at boundary?	`readelf -Ws`, disassembly prologue/epilogue	SysV AMD64 calling convention intact

H.10 Optional Variant — PGO + LTO

Rebuild with profile feedback:

```
g++ -O3 -flto -fprofile-generate -march=native -o app.gen case.cpp driver.cpp
./app.gen
g++ -O3 -flto -fprofile-use -fno-peel-loops -march=native -o app.pgo case.cpp
↪  driver.cpp
```

What should change:

- **Block layout**: hot path reordered; cold exit checks sink out of I-cache footprint.
- **Inlining**: cross-TU inlining (if multiple TUs).
- **Vector epilogue**: may be simplified if profile indicates favorable `n % VL`.

Validate with `-fopt-info-vec -fopt-info-inline` and compare `perf stat` deltas.

H.11 Minimal Troubleshooting Guide

- **Vectorization refused**: Check `vect` dump for "dependence" or "misaligned/unknown step". Enforce alignment with `assume_aligned` or adjust data layout.

- **High branch-misses**: Ensure loop is branchless; guard conditions hoisted out. Consider masking rather than branching for special cases.

- **Low IPC with high L1 misses**: Confirm contiguous layout and that prefetchers can engage; avoid gathers; consider SoA refactor.

- **Spills visible**: Reduce live range pressure (split accumulators), or limit width (`-mprefer-vector-width=256`) where AVX-512 causes register pressure.

H.12 Outcome

This case study ties together:

- **AST → GIMPLE/SSA → RTL → Assembly**: each stage inspected and matched to the final code.

- **Optimization reasoning**: explicit acceptance evidenced by `-fopt-info-*` and dumps.

- **Measured reality**: `perf` counters confirm that structural changes improve microarchitectural utilization.

- **ABI invariants**: symbol forms and calling conventions verified to remain stable.

Use this template to evaluate any performance-critical kernel: keep the dumps, the annotated disassembly, and the perf reports together. The correspondence between **compiler theory**, **binary facts**, and **runtime behavior** is the basis for reliable, high-performance C++ system construction on GCC for Linux x86-64.

Appendix I - Experimental and Research Extensions

This appendix summarizes advanced mechanisms for extending, instrumenting, and experimentally modifying the GNU compilation pipeline. These topics are intended for compiler researchers, performance engineers, and system architects who require beyond-standard transformations, cross-toolchain interoperability, or runtime compilation strategies. All material reflects GCC behavior and capabilities available in post-2020 versions (GCC 10+ and libgccjit 10+), with relevance to Linux x86-64 environments.

I.1 GCC Plugin Interface and Custom Optimization Passes

GCC exposes a plugin API that allows external modules to register new passes, inspect or transform intermediate representations, and participate in compilation workflows. Plugins operate on **GIMPLE** or **RTL**, depending on the insertion point, and are integrated via dynamic loading (`-fplugin=<path>`).
Key components:

- Registration via `plugin_init(struct plugin_name_args*, struct plugin_gcc_version*)`.
- Pass insertion with `register_callback` targeting `PLUGIN_PASS_MANAGER_SETUP`.
- IR hooks to traverse:
  - GIMPLE statements (`gimple_stmt_iterator`, `FOR_EACH_BB_FN`),
  - SSA use-def chains (`ssa_name`, `def->use chain`),
  - RTL insns (`FOR_EACH_INSN`).

Use cases:

- Automatic insertion of diagnostic assertions for correctness validation.
- Experimental optimization strategies (e.g., domain-specific strength reduction).
- IR instrumentation (e.g., branch frequency logging).

Plugins must respect GCC's phase ordering constraints: transformations must maintain SSA validity and dominance relationships when applied at the GIMPLE level, and register constraints when operating on RTL.

I.2 Cross-Toolchain Comparison via LLVM Interchange

GCC can interoperate with LLVM at the **LTO** level via:

- Emission of LLVM bitcode for selected units,
- Symbol table and type metadata correlation via DWARF,
- Comparative passes for code-generation or scheduling differences.

Research workflow:

1. Build shared input corpus with stable command-line configuration.
2. Compile with GCC (`-flto`) and LLVM (`-flto` or `-emit-llvm`).
3. Compare:
    - GIMPLE vs LLVM IR structure,
    - Inlining heuristics,
    - Loop and vectorization transformations,
    - Register allocation outcomes via annotated disassembly.

Objective:

- Identify optimization inefficiencies or structural advantages in either pipeline,
- Selectively port pass logic to GCC plugins or backend tunings.

I.3 Source-Based Coverage Instrumentation

GCC supports non-intrusive profiling through:

- `-fprofile-generate` / `-fprofile-use`
- `-fprofile-update=atomic` for thread-safe counters,
- `-fprofile-values` for indirect-call target histograms.

For precise research measurement:

- Use coverage to derive **branch probability**, **loop trip distributions**, and **call frequency**.
- Extract histograms from `.gcda` using `gcov` or direct binary parsing.

This enables:

- Cost model tuning for vectorization and inlining,
- Feedback-driven function layout,
- Adaptive specialization strategies compatible with the static binary ABI.

I.4 Automated Code Layout Optimization via Block Reordering

Post-2020 GCC includes improved support for function reordering and hot/cold text partitioning, influenced by PGO data. Researchers may augment this by:

- Modifying block reordering passes to align hot paths with I-cache residency constraints,
- Coalescing contiguous hot regions into `.text.hot`,
- Pushing cold blocks to `.text.unlikely` for reduced branch pressure.

Evaluation metrics:

- Front-end stall cycles (idq_uops_not_delivered),
- Instruction-cache misses (icache.misses),
- BTB mispredict frequency under highly biased calls.

This strategy is architecture-sensitive and should be validated per microarchitecture class.

I.5 Runtime Compilation Using libgccjit

`libgccjit` enables dynamic generation of machine code using GCC's code generator. Instead of interpreting IR, it constructs GCC-internal representations programmatically and emits executable code that adheres to:

- System V AMD64 ABI,
- Standard alignment and calling conventions,

- GCC's backend instruction scheduler.

Use cases:

- JIT specialization based on runtime data distribution,
- Runtime code synthesis in simulation frameworks,
- Micro-benchmarks comparing static vs dynamic optimization outcomes.

Constraints:

- No access to full midend pipeline; transformations available are lower-level.
- Intended for JIT research, not general-purpose compilation replacement.

I.6 Research Workflow Integration Model

Objective	**Appropriate Mechanism**	**Expected Output**
Inspect/Modify IR	GCC Plugin (GIMPLE/RTL)	Selective rewriting of transformation passes
Cross-compiler benchmarking	LLVM Interop + disassembly correlation	Structural optimization comparison
Dynamic performance feedback loops	`-fprofile-*` instrumentation + PGO	Performance-informed inlining and block layout
Code layout optimization	Hot/cold partitioning, custom linker scripts	I-cache residence improvement

Objective	Appropriate Mechanism	Expected Output
Runtime specialization	`libgccjit`	On-demand optimized code generation

I.7 Outcome

This appendix establishes research interfaces into the GCC toolchain that:

- Preserve ABI correctness,
- Allow controlled experimentation on IR structures,
- Support cross-backend comparative evaluation,
- Enable runtime adaptation without breaking system linkage conventions.

These mechanisms provide the foundation for:

- Compiler performance research,
- Architecture-aware workload specialization,
- Automated tuning of large C++ systems.

They are suitable for graduate-level study, compiler design experimentation, and high-end performance engineering.

References

This book draws from authoritative and publicly documented specifications, compiler implementation sources, operating system standards, and microarchitectural design literature. The references below represent stable, versioned, and technically primary sources that define the behaviors, invariants, and semantics discussed throughout the text. They are listed to enable verification, deeper study, and long-term technical continuity.

The references are organized by conceptual domain rather than by chapter, reflecting the layered structure of the compilation and execution environment.

Language and Semantic Foundations

1. **ISO/IEC 14882 — Programming Language C++ Standard** (C++20 and later drafts)
   Defines core language semantics, name lookup rules, template instantiation, constexpr evaluation, object lifetime, and concurrency primitives.

2. **ISO/IEC 9899 — Programming Language C Standard** (C17 / C23)
   Serves as foundational reference for C++'s low-level and memory semantics.

3. **C++ Core Guidelines (Stroustrup, Sutter, and WG21 contributors)**

Describes best practices in type safety, resource management, concurrency, and API stability, consistent with modern C++ compilation behavior.

Compiler Internals and Intermediate Representations

1. **GCC Internals Manual (GCC 10 and later)**
   Formal reference for GCC front-end lowering, GIMPLE IR, SSA construction, RTL definition, pass sequencing, and backend instruction emission.

2. **GIMPLE and SSA Form Framework Specification (GCC Developer Documentation)**
   Defines correctness rules for SSA dominance, phi node placement, range propagation, and GIMPLE canonicalization invariants.

3. **RTL Machine Description Language Reference**
   Documents instruction patterns, operand constraints, register classes, and target backend code generation rules.

ABI and Binary Interface Specifications

1. **System V AMD64 ABI**
   Defines calling conventions, stack alignment, argument passing rules, VTable layout, exception propagation structures, and typeinfo object shape.

2. **Itanium C++ ABI**
   Specifies class layout, RTTI format, virtual dispatch tables, construction vtables, thunks, and exception personality functions.

3. **ELF Object File Format Specification**
   Describes section layout, program headers, relocation records, and dynamic linking metadata structures.

4. **DWARF Debugging Information Format**
   Defines symbol tables, call frame information (CFI), unwind tables, and line number mappings for debugging and exception unwinding.

Runtime and Operating System Interfaces

1. **glibc Runtime Behavior and Loader Internals (dynamic loader and startup code)**
   Reference for GOT/PLT resolution, dynamic relocation, TLS models, constructor/destructor registration, and process startup sequence.

2. **Linux Kernel System Call ABI Documentation**
   Defines user–kernel transition models and calling conventions for syscall entry and return flows.

Microarchitectural and Performance Analysis Sources

1. **Intel 64 and IA-32 Architectures Optimization Reference Manual**
   Describes pipeline topology, execution port mapping, µop scheduling, instruction throughput/latency, branch prediction behavior, and memory hierarchy costs.

2. **AMD Architecture Programmer's Manual (Zen µArch families)**
   Reference for cache hierarchies, TLB behavior, SIMD execution characteristics, and port utilization constraints.

3. **perf Event Reference for Linux Performance Counters**
   Defines hardware counter groups, stall classification models, IPC computation, and event correlation for pipeline attribution.

Toolchain, Debugging, and Profiling

1. **GDB Internals and Python API Reference**
   Covers frame reconstruction, symbol resolution, reverse execution, automated pretty-printing, and runtime heap state inspection.

2. **binutils (objdump, readelf, nm, ld) Technical Documentation**
   Defines binary inspection workflow and static linking control.

Research and Extension Interfaces

1. **GCC Plugin API Specification**
   Describes registration hooks, GIMPLE passes, IR transformations, and custom analysis integration.

2. **libgccjit JIT Compilation Interface**
   Enables runtime generation of GIMPLE and machine code for dynamic and adaptive workloads.

20.6 Purpose of Reference Structure

This reference list is intentionally **primary-source oriented**:

- Every cited work defines behavior rather than describes or interprets it.

- All specifications are stable or versioned to ensure reproducibility.
- The list avoids tutorial and secondary commentary sources to maintain technical exactness.

The reader is encouraged to use these references not as introductory material, but as precise verification anchors when inspecting compiler output, debugging complex runtime behavior, or designing performance-critical system components.