Foundational Problems with Compilers and Operating Systems

Brent Hartshorn July 15th, 2025

brenthartshorn@proton.me

Abstract

This paper addresses the fundamental inefficiencies in modern compiler and operating system designs, particularly as they impact high-performance Linux server environments. We propose a radical re-evaluation of core architectural choices, advocating for indexed and compressed stack pointers, and a novel thread management scheme that locks threads to specific cores for ultra-fast context switching. The discussion extends to the utility of direct memory access for specialized server applications, challenging traditional security paradigms and proposing a Python-to-ASM toolchain for enhanced control and optimization. We further highlight the performance advantages of ARM and RISC-V architectures over Intel/AMD, specifically their suitability for advanced threading models. Finally, we examine critical overlooked performance bottlenecks within the Linux kernel concerning SIMD register management and propose enhancements to the ELF file format to optimize process execution. By implementing these foundational shifts, organizations can achieve a huge improvement in server performance or a huge reduction in operational costs.

Introduction

The continuous demand for faster and more efficient server infrastructure necessitates a critical examination of the foundational technologies underpinning our computing environments: compilers and operating systems. While significant progress has been made in hardware capabilities, the software layers that manage and execute applications often harbor inefficiencies that bottleneck true performance potential. This paper argues that merely optimizing applications is insufficient; a paradigm shift in how compilers and operating systems are designed and interact is required to unlock unprecedented levels of speed and cost efficiency, particularly in high-throughput Linux server deployments.

The core of our argument centers on challenging long-held design assumptions that, while offering broad compatibility and general-purpose utility, inadvertently introduce substantial overhead. We delve into specific areas such as the design of stack pointers, the management of threads and context switching, and the trade-offs between security layers and raw performance. Furthermore, we explore how development toolchains and even executable file formats can be re-imagined to provide developers with greater control and optimize for the unique demands of server-side software. By addressing these foundational problems, from the very first instruction executed by an operating system to the final compiled binary, we can pave the way for a new era of server performance.

Rethinking Stack Management and Threading for Performance

From the very outset of an operating system's design, the traditional approach to stack pointers introduces inefficiencies. Instead of a linear, unconstrained stack pointer, an optimized design would feature an indexed stack pointer. This pointer would reference memory offsets at fixed intervals, for example, 16-byte increments, and only allow increments of 16 bytes. This strategic alignment enables the compression of stack pointers, potentially allowing 64-bit addresses to be represented within 32 bits or even less. Consequently, a single 64-bit register could effectively house two such compressed stack pointers, significantly improving memory efficiency and access speed.

Furthermore, the entire architecture, from the operating system kernel to the compiler, should be re-envisioned to prioritize multiple threads per core. These threads should be explicitly locked to their respective cores and organized into "left" and "right" thread groups. This design facilitates highly efficient context switching by minimizing the need to save and restore registers. By dedicating specific register sets to each thread group, a context switch between threads within the same core would primarily involve only switching the program counter and a minimal amount of core-local state, drastically reducing overhead.

The Case for Direct Memory Access and Simplified Toolchains

For developers building high-performance server software, particularly in environments like containers or unikernels where the software's behavior and interactions are tightly controlled and understood, traditional security mechanisms, such as virtual memory, can become a performance bottleneck. In such scenarios, where developers "know what they are doing," direct memory access offers substantial speed gains by eliminating layers of abstraction and overhead. This bypasses the Translation Lookaside Buffers (TLBs) and page tables, which are a major source of latency in virtual memory systems, and eradicates page faults.

While such an approach would typically lead to severe memory fragmentation, this can be mitigated in a highly controlled environment. If a Just-In-Time (JIT) C compiler is integrated directly into the kernel, and this compiler has foreknowledge of all programs intended to run for a given session, it could pre-allocate disjoint physical memory regions for each program. This compiler-kernel collaboration would eliminate dynamic memory fragmentation, further enhancing the benefits of direct memory access by ensuring optimal physical memory layout and potentially improving cache locality.

The development toolchain itself should also be re-evaluated. Instead of relying on complex languages and compilation processes, a more direct approach is advocated. A Python codebase can serve as a powerful meta-programming layer, generating all necessary artifacts, including direct assembly code, linker scripts, and Makefiles. This approach simplifies the development of compilers and build tools, allowing for rapid iteration and highly optimized code generation tailored to specific hardware. This contrasts sharply with languages like C++, which, despite their widespread use, can hinder the ease of parsing and direct transformation into assembly.

The Role of Hardware in Performance

The choice of server-side hardware significantly impacts networking performance, which is heavily influenced by the speed of thread and process context switching at the operating system level. Traditional Intel and AMD architectures, with their 16 registers per core, are not optimal for advanced context switching techniques. In contrast, ARM and RISC-V architectures, boasting 32 registers, allow for a "left and right" grouping of threads. This enables super-fast context switching and significantly accelerates inter-thread communication by locking threads to the same core and utilizing shared registers, effectively providing lock-free atomic safe memory. Therefore, for optimal server performance, migrating from Intel or AMD to ARM or RISC-V hardware can yield substantial speed improvements.

Linux Kernel Overhead with SIMD Registers

A significant, yet often overlooked, performance impediment within the Linux kernel pertains to its handling of Single Instruction, Multiple Data (SIMD) registers. When the kernel detects that a process utilizes SIMD registers (such as those for SSE, AVX, or particularly AVX-512), it incurs a substantial overhead by meticulously saving and restoring the entire state of these registers during every context switch for that process. This behavior, designed to maintain process isolation and ensure correct execution across different contexts, can lead to a considerable performance hit, especially with wide SIMD registers like AVX-512 which have a large state. For high-performance server applications that frequently use SIMD instructions, this automatic saving and restoring becomes a major source of latency, drastically reducing the benefits derived from parallel processing. Optimizing or selectively disabling this behavior for trusted processes in controlled server environments could yield significant performance gains.

Security and Language Design

While languages like Rust invest heavily in memory safety and security features, their efficacy is often constrained by their reliance on the underlying operating system for fundamental security mechanisms like virtual memory. If true, uncompromising security is the goal, a holistic approach is necessary: designing the language, compiler, and operating system in unison. This is a concept explored by figures like Terry Davis with TempleOS, where the operating system and its development environment were tightly integrated and designed from the ground up for a specific vision. For high-performance server environments, particularly with applications deployed in containers or unikernels where the execution environment is highly controlled, the overhead introduced by traditional OS-level security layers can be counterproductive. In such specialized contexts, where developers maintain strict control over the codebase and execution environment, direct memory access can be embraced, bypassing layers that introduce performance penalties. Ultimately, for maximum control and performance in these scenarios, a Python layer capable of translating directly to assembly offers the most streamlined path.

Limitations of the ELF File Format

The Executable and Linkable Format (ELF), a standard file format for executables, object code, shared libraries, and core dumps, while versatile, lacks certain crucial features for highly optimized server environments. Specifically, the current ELF specification does not provide a mechanism for an executable to declare its exact register usage or, more generally, which specific CPU features or register sets it will exclusively employ. This omission forces the operating system kernel to make conservative assumptions about register state during context switches, leading to the aforementioned overhead of saving and restoring all possible registers, including large SIMD register sets, even if a process only utilizes a small subset. An enhanced ELF format that allows explicit declaration of required register groups would enable the kernel to perform more intelligent and minimal context saving, thereby significantly reducing overhead and boosting performance for processes with well-defined register requirements.

Strategic Investments for Performance Enhancement

To achieve these significant performance and cost improvements, several key areas require strategic investment:

1. **Linux Kernel Optimization**: The Linux Kernel is often the primary server-side bottleneck. Customizing and optimizing the kernel for the specific hardware in use is crucial. This includes addressing existing performance-killing compromises within the kernel's source code, often related to hardware compatibility.
2. **Compiler Optimization (GCC/Clang/LLVM)**: The compiler is a critical component that also needs optimization. Extending the C language for lower-level control and integrating custom assembly optimizations, despite conventional wisdom, can unlock further performance gains.
3. **Harnessing RISC-V/ARM Architectures**: Modifying compilers like GCC to take advantage of the larger register files (32 registers) in RISC-V and ARM architectures enables the implementation of "left and right" thread groups. This allows context switching between these threads on a single core to occur with minimal saving and restoring of registers, leading to drastically reduced context switch overhead, improved cache locality, and higher throughput for parallel workloads.

By adopting these foundational shifts in design philosophy and making targeted investments, organizations running extensive Linux server infrastructures can unlock substantial performance improvements and cost efficiencies, potentially achieving a huge increase in speed or a huge decrease in operational costs.

References:

[1] Fair Scheduling for AVX2 and AVX-512 Workloads – USENIX

https://www.usenix.org/system/files/atc21-gottschlag.pdf

[2] Automatic Core Specialization for AVX-512 Applications
https://os.itec.kit.edu/downloads/Automatic_Core_Specialiszation_for_AVX_512_Application.pdf