Avoiding system call overhead via dedicated user and kernel CPUs

Josh Triplett
josh@joshtripplett.org
Portland State University

Philip W. Howard
pwh@cs.pdx.edu
Portland State University

Eric Wheeler
ewheeler@ewheeler.net
Portland State University

Jonathan Walpole
walpole@cs.pdx.edu
Portland State University

Abstract

We show that we can avoid syscall overhead by dedicating a separate CPU to execute syscalls on behalf of a userspace process. Our prototype on Linux performs syscalls 70% faster than the standard syscall interface.

1 Introduction

Freed from the diminishing returns of increasing CPU speed, hardware designers have resumed their exponential growth by providing an ever-increasing number of cores. Intel’s latest research chip, the Single-chip Cloud Computer (SCC), sports 48 cores on a single chip. The Sun UltraSparc T2 handles 64 hardware threads at one time. The growing number of cores provides the opportunity to re-think some of the basic premises of operating system design.

The earliest operating systems used batch scheduling: individual jobs started and ran to completion without interruption. The advent of multiprocess scheduling allows increased interactivity, multi-user systems, and potentially better saturation of available resources. Multiprocess scheduling provides the illusion that each task has a dedicated CPU, but maintaining this illusion requires expensive context switches every time a new thread starts running. Protecting the OS kernel itself requires expensive privilege level switches. These operations incur high-cost virtual memory manipulations, TLB misses, and multi-level cache misses because different processes typically reference disjoint parts of physical memory.

Given enough processing cores, many workloads will find themselves with more cores than processes, turning multiprocess scheduling into a source of overhead with little gain. These workloads may split work over multiple cores, but this incurs communication and scheduling overhead. With this surfeit of cores, we could take a cue from the batch schedulers by minimizing unnecessary context switches in a process that has a core to itself. Even with this optimization, processes must pay a penalty for each syscall to maintain operating system protection.

We propose rethinking the division between userspace and kernel space to eliminate the overhead of system calls. Rather than dedicating only a single core to a process, we dedicate two: one to run the user-mode process, and one to perform system calls and the associated kernel-mode computation. Each core can keep its context “warm” on the CPU, and need not switch contexts for each syscall.

This design does not require a rewrite of the OS; only the syscall interface needs to change. Rather than invoking a call gate that traps to the OS—incurring multiple penalties—the syscall interface packages the syscall and parameters as a message and sends them to the kernel CPU. OS and user code can benefit without changes.
2 Prototype

As an initial prototype, we implemented a new system call interface for the Linux kernel. A process making use of this interface creates a new thread to call our new system call: sys\_batch. This system call communicates with the original thread via a shared-memory buffer, and never returns to the calling thread until the entire process exits. The original thread performs system calls by writing them into the shared memory buffer, and sys\_batch calls the corresponding kernel function and writes the result to the shared memory buffer.

Our initial prototype executes system calls synchronously, with only one system call outstanding at a time. This approach does not provide any opportunity for pipelining, and thus any performance benefits come entirely from avoided system call overhead.

We compared the performance of our sys\_batch approach to the standard Linux system call interface, as provided by the syscall() function (bypassing the usual libc wrapper functions). Our test system used an Intel Core 2 Duo, running Linux for the x86-64 architecture, for which the standard system call interface uses the dedicated hardware instruction “syscall”. To control for timing overhead and background OS activity, we timed multiple runs of millions of syscalls.

3 Performance Results

On our test system, a simple getpid() system call took about 120 nanoseconds via the standard system call interface. Using sys\_batch reduced this to 70 nanoseconds per call. More complex system calls increased both timings by a constant factor, but did not eliminate the delta between them.

These results show that communication between a dedicated user and kernel CPU can outperform syscall overhead, even with optimized hardware instructions for performing syscalls.

4 Future Work

Our sys\_batch interface can run syscalls asynchronously, allowing userspace processes to pipeline multiple syscalls and wait for the results only when needed. This significantly reduces latency, but requires adaptations to existing applications to defer requesting results. System calls which return useful data will necessitate larger adaptations, but for the large class of system calls which return only an error code, asynchronous error handling would require minimal adaptation.

Our initial prototype provided a single kernel process for a single userspace process. If the kernel can switch between userspace process “contexts” without changing privilege level, sys\_batch could allow one kernel process to serve many userspace processes, consolidating kernel data structures onto fewer CPUs and reducing synchronization overhead.

Conversely, a syscall-bound user process could spawn many kernel threads to service its syscalls, and process results as they become available, maintaining whatever ordering constraints it requires by sending syscalls to the same kernel thread. This may enable a new class of event-driven applications or userspace threading techniques.

Real-time applications care deeply about “jitter” caused by kernel-mode execution, even when that execution occurs on their behalf. Real-time scheduling can avoid interrupting a real-time process until it explicitly blocks, but that process may still need to interact with the system. sys\_batch would allow a real-time process to run entirely uninterrupted on a core, completely eliminating jitter, while still allowing that process to make system calls.

Existing research provides highly optimized techniques for remote procedure calls and messaging, most of which will apply directly to the syscall messages used by sys\_batch. With sys\_batch, syscall speed becomes limited only by communication latency and throughput between CPUs, with the privilege level switch no longer an issue.