Exploiting Parallelism in OS Kernel System Calls

SuperUROP Proposal by Varun Mohan

Faculty Supervisor: Frans Kaashoek

Abstract

System calls, the standard procedure for requesting resources from the kernel, are renowned for being expensive in terms of performance. Moreover, even though kernels have been optimized for multicore settings, they have not been able to reduce the performance penalties of system calls since they run system calls on only a single CPU. In this research, we will focus on trying to parallelize some of the more compute-intensive system calls like "fork" and "exec" to reduce the time spent in the kernel. We will be working with the Biscuit kernel written in Go, which has built-in support for transient threads, and will also analyze the performance overhead of spawning threads within a single system call. Ultimately, the goal is for single threaded kernel-intensive applications to have a significant speedup in multicore environments.

good mentioning broader problem and gap in existing research

Problem Statement

Modern operating systems kernels must be written in a way that allows them to scale with the number of CPU cores. However, currently, operating systems do not take advantage of all the system's cores to speedup a singular single-threaded application. Specifically, whenever a users pace process requests resources from the kernel via a system call, the kernel usually executes the system call on a single core. If instead the kernel were able to execute these system calls concurrently across multiple cores, this would present a significant performance improvement for applications that spend a lot of time in the kernel. In addition, system calls are usually very expensive since they can require multiple context switches, which consists of many

citation here?

citation here?

process registers being stored and subsequently loaded from memory. Therefore, limiting the amount of time spent in the kernel performing a system call would be extremely useful for kernel-intensive applications and would provide an immediate speedup without any code changes.

Good job explaining problem and gap in knowledge

In this research, we will be exploring this new approach of parallelizing singular system calls in the Biscuit kernel, developed by researchers at the PDOS group. Biscuit is written in Go, a language with built-in concurrency primitives, making it much easier to parallelize these system calls relative to a kernel written in the C programming language. Furthermore, we plan to parallelize computation intensive system calls like “fork” and “exec” in the Biscuit kernel since otherwise the performance overhead of spawning a thread inside the system call would not be justified. The end result of the study is hopefully that single-threaded applications like Redis can achieve large speedups by exploiting all CPU cores while in the kernel. In the process, we will also analyze the cost associated with spawning threads within a system call and see if there are ways of modifying the Go run-time to make it more suitable for the kernel.

citation here?

Related Work

looks like a lot of background on problem in general, not necessarily the related research

Modern operating system kernels like Linux are written in the C programming language, which does not provide concurrency primitives as well as the simple mechanisms to spawn transient threads. Furthermore, this leads to kernel using very complicated mechanisms for managing concurrent tasks while utilizing multiple cores. The main reason for the difficulty involved in thread creation arises from the lack of garbage collection in C. Specifically, in the absence of garbage collection, it becomes very difficult to manually free objects that are shared across multiple threads. For these reasons, there has been a shift towards kernels written in high-level languages (HLL), which provide garbage collection as well as easy-to-use threading

??

support, thereby make it easier to reason about kernel and write bug-free code [references]. However, current kernels written in a HLL are usually either specialized like KaffeOS, which

Good specifics. But how does this relate?

can only run JVM based processes, or are proof-of-concepts that do not place much focus on performance [1, 2].

Furthermore, researchers at the PDOS group have implemented the performant kernel Biscuit using the HLL Go, a type-safe garbage collected language with transient thread support. Since the Go run-time is supposed to be executed as a user space process that allocates memory from an underlying kernel, the Biscuit kernel has a "shim" layer that simulates these functions on physical hardware and sits right below the run-time as in Figure 1. The Go run-time then acts as a scheduler for all kernel threads and user space processes in Biscuit. The Biscuit kernel has shown high performance on a single core machine, achieving within 75% of Linux on two kernel intensive application, Redis and simple mail delivery. In fact, most of the differences can be

citation here?

attributed to additional optimizations in Linux with only a 5 – 10% slowdown due to garbage

citation here? give a specific

collection. Moreover, it is possible to write a kernel in a HLL that does not sacrifice performance. In addition, since the Go run-time provides built-in lightweight threads called goroutines, Biscuit has even more opportunities to improve performance through added parallelism.

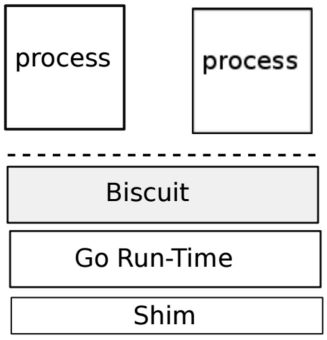

Figure 1: Biscuit's architecture

The topic of optimizing system calls is of high interest since they pose significant overhead for applications that frequently request resources from the kernel. One study created FlexSC, an implementation of exception-less system calls or system calls that can be executed asynchronously, taking advantage of system call batching as well as CPU core specialization [3]. With this method, they were able to optimize many kernel-intensive applications like MySQL and Apache. Another approach is through system call clustering, which uses profiling to group multiple system calls into a single system call, reducing the number of user/kernel context switches [4]. However, both these approaches do not attempt to improve the performance of the underlying system call function but instead focus on lessening the effects of synchronous context switches on user space processes. Moreover, to our knowledge, the idea of parallelizing the implementation of singular system call functions has not been researched in the past and could potentially provide major improvements.

citation here?

I would have liked to see more examples of research related. There were only a few mentioned

Technical Approach

Most system calls do not involve computationally intensive sections in the kernel, including those that block on I/O. For this research, we will be focusing on the system calls that execute many instructions while in the kernel. Moreover, for lightweight system calls, the overhead of spawning kernel threads will outweigh the performance benefits. Furthermore, we have decided to consider system calls like “fork,” “exec,” “mmap,” “read,” and “write.”

Specifically, the “fork” system call, which creates a copy of a process, needs to copy over the process’ entire page table which could have millions of mappings, assuming a 64-bit address space. We expect that parallelizing this copying procedure will produce improvements. Similarly, for the “exec” and “mmap” system calls, copying over a process image or file into memory could be sped up if done concurrently across multiple cores. Finally, we can parallelize

Very clear explanation. Good job

system calls like “read” and “write” as in Figure 2, wherein each core is tasked with copying over a disjoint set of pages from the buffer cache to a user space buffer.

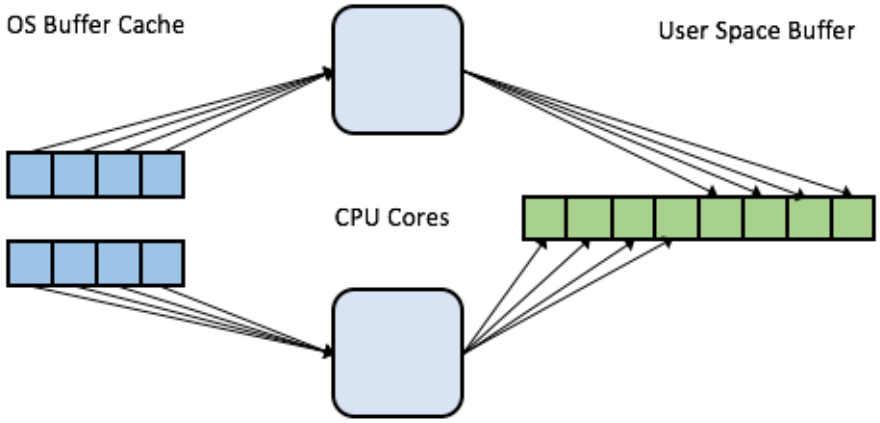

Figure 2: Parallelized “read” system call

For the most part, we expect for sufficiently large data copies in the above system calls that parallelization will provide significant speedup. However, it is not clear how much overhead the Go runtime will impose on performance while spawning threads. We will therefore also study the Go run-time schedule and determine whether it needs to be modified to be more suitable for the kernel.

Maybe go into a bit more depth here. It's pretty sparse.
Also reference previous work

Evaluation Metrics

The efficiency of the singular parallelized system calls will be evaluated based on their speed compared to the speed of a regular serial implementations that only use a single core. Furthermore, we will first stress test the system by applying the parallel and serial system calls to cases where a large amount of data needs to be copied. In addition, we will also look at the difference in performance for real kernel-intensive applications like Redis that read, write, and map large chunks of memory. Note that our parallel system calls will invoke an eager approach, For instance, in the case of the “mmap” system call, the address space is mapped synchronously with respect to the system call. Linux, however, applies a lazy approach wherein the address

What type of system configurations will you use. Surely that will affect performance

mappings are generated during an actual page fault. Furthermore, we will also evaluate the difference in performance between our parallelized eager system calls against their serial lazy counterparts.

In addition, we will also evaluate the overhead of spawning a lightweight thread in Go. Specifically, we can quantitatively determine the time it takes to spawn as well as rejoin the original thread. In the process, we may find ways to improve the Go run-time's scheduler specifically for the kernel.

Proposed Timeline

- October till December – decide the set of system calls that will be parallelized and implement their parallelized counterparts.
- February & March – benchmark the performance of the parallelized system calls against their serial counterparts as well as lazy implementations for kernel intensive applications.
- April & May – fine tuning of parameters to determine when parallel system calls provide speedup and explore other possibilities of exploiting parallelism that are uniquely easier to implement using Go. We will also try to modify the go runtime so that spawning of transient threads is more ideal in a kernel setting.

Conclusion

Initial testing has shown that parallelizing the system call "fork" does in fact provide significant performance improvements for processes with large address spaces. However, it is not obvious whether this will be the case for all of the outlined system calls or whether a trick like lazy initialization will be able to overshadow the benefits of this optimization. Hopefully, this novel approach yields a new way that the kernel can enhance performance for a single

You're brining up new stuff that wasn't mentioned earlier.
That could have been mentioned in challenges

process running in a multicore environment. In addition, the study will also provide some insight into overhead of transient threads in a kernel environment.

Works Cited

1. G. Back and W. C. Hsieh. The KaffeOS Java runtime system. *ACM Trans. on Prog. Lang. and Sys.*, 27(4): pages 583–630, 2005.
2. Redox - Your Next(Gen) OS. http://www.redox-os.org/.
3. L. Soares and M. Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. *In Proceedings of the 9th Symposium on Operating System Design and Implementation* (OSDI '10), pages 33–46, 2010.
4. M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting. System call clustering: A profile directed optimization technique. Technical report, The University of Arizona, May 2003.