OS Execution on Multi-Cores: Is Out-Sourcing Worthwhile?

David Nellans, Rajeev Balasubramonian, Erik Brunvand

School of Computing, University of Utah
Salt Lake City, Utah 84112
{dnellans, rajeev, elb}@cs.utah.edu

Abstract

Large-scale multi-core chips open up the possibility of implementing heterogeneous cores on a single chip, where some cores can be customized to execute common code patterns. The operating system is an example of a common code pattern that is constantly executing on every processor. It is therefore a prime candidate for core customization. Recent work has begun to explore this possibility, where some fraction of system calls and other OS functionality is off-loaded to a separate special-purpose core. Studies have shown that this can improve overall system performance and power consumption. However, our explorations in this arena reveal that the primary benefits of off-loading can be captured with alternative mechanisms that eliminate the negative effects of off-loading. This position paper articulates this alternative mechanism with initial results that demonstrate promise.

1. Motivation

In the era of plentiful transistor budgets, it is expected that processors will accommodate tens to hundreds of processing cores. With processing cores no longer being a luxury, we can consider dedicating some on-chip cores for common applications. The customization of these cores can allow such applications to execute faster and more power-efficiently. One such common application is of course the operating system: it executes on every processor and is frequently invoked either by applications or simply to perform system-level book-keeping. The operating system is an especially important target because several studies (Nellans et al. 2005; Li and John 2006; Agarwal et al. 1988; Chen and Bershad 1993) have shown that the past decade of microarchitectural innovations have done little to boost the performance of OS execution. This is attributed to many factors: OS calls are short, have hard-to-predict branches, have little instruction-level parallelism (ILP), and suffer from cache interference effects. It can also be argued that current high-performance cores are over-provisioned for OS execution (for example, floating-point units, large reorder buffer, large issue width) and are hence highly inefficient in terms of power consumption. Studies (Redstone et al. 2000; Nellans et al. 2005; Chakraborty et al. 2006) have shown that operating system code constitutes a dominant portion of many important workloads such as webservers, databases, and middleware systems. Hence, optimization of OS execution (perhaps with a customized core) has the potential to dramatically impact overall system performance and power.

Some research groups (Mogul et al. 2008; Chakraborty et al. 2006; Nellans et al. 2005) believe in the potential of core customization within multi-cores to improve OS efficiency. Our own work attempted to test this hypothesis. Our findings broadly agree with this premise, but we identify the key factors that make off-loading desirable. We propose an alternative approach that incorporates these factors within each core to eliminate the negative effects of off-loading.

2. One Problem - Two Existing Approaches

Until recently, chip multiprocessors were not common and approaches to operating system support were limited to *intra-core* solutions. Li and John (Li and John 2006) focused on reducing the total power consumption of a processor by reconfiguring and augmenting microarchitectural structures during operating system execution to improve energy efficiency. They identify aliasing in branch predictors as a major source of conflicts between user and OS codes and propose hardware extensions that decrease the aliasing. They find that allowing L1 caches to enter lower power modes and limiting power hungry microarchitectural features such as the re-order buffer and instruction fetch/issue width during OS execution can yield favorable energy-delay products. Their results show that an aggressively designed machine which can dynamically reconfigure into a modest 2-issue machine during OS execution can achieve performance within 10% of a 6-issue machine, while consuming less than 50% of the power.

A second alternative approach takes advantage of the availability of many cores with relatively low inter-core latencies. Chakraborty et al. (Chakraborty et al. 2006) focus on migrating system call execution from multiple cores onto statically allocated OS cores via "Computation Spreading". Targeting an 8-core symmetric CMP design, they show a 25% increase in ILP for some OS intensive workloads.

Similar in concept, but focusing on energy efficiency, Mogul et al. (Mogul et al. 2008) couple an energy-efficient core (based on an EV4 design) with an aggressive core (based on an EV6 design). They migrate long running system calls to the power-efficient core (the *OS core*) and power down the aggressive core during this off-load. This achieves substantially improved energy efficiency over a uni-processor design but at a detriment to total throughput.

3. Non-Interference without Off-Load

The off-loading approach proposed by Chakraborty et al. attributes increased performance to data-reuse in caches because similar system calls are co-scheduled on a single core. In our attempts to design an effective off-load mechanism, we too discovered that reduced OS-user interference in caches can provide a significant performance boost. However, we observed that the overall performance improvements can be relatively low because of the following reasons:

- The off-load mechanisms of Chakraborty et al. and Mogul et al. rely on software support process migration that incurs overheads of many thousand cycles on every off-load. This makes off-loading worthwhile only for system calls that execute more than 10,000 instructions. Unfortunately, even in OS-intensive workloads such as Apache and SPECjbb, only about 2% of all system calls would qualify for off-load currently. Low-latency off-load mechanisms, an area of active research (Brown and

Tullsen 2008; Strong et al. 2009), will be required to fully take advantage of the OS core's cache for most system calls;

- OS execution often shares a significant amount of data with user threads because of its handling of privileged I/O mechanisms. If the OS syscall is off-loaded to its own core, such shared data will not only have to be replicated in two caches, but access to this data will incur expensive cache coherence misses. In such cases, making the OS and user threads access different (and relatively distant) cache structures is an impediment to high performance.
- Off-loading will eventually yield marginal returns (or potential slowdowns) as the number of cores serviced by a single OS core increases. In other words, a single OS core will not scale well at future technologies. For OS-intensive applications, our experiments showed poor scalability even when an OS core serviced requests from two application threads.

Based on the above observations, we believe that the cache segregation effects of off-loading must be provided without moving OS syscall execution to a different core. We therefore propose a cache structure on every core that can handle OS working sets. Just as we have become accustomed to L1-instruction and L1-data caches in modern processors, it is perhaps now time to consider a specialized OS cache for future processors. For the rest of this discussion, we will assume that our proposed processor incorporates a *user-cache* and an *OS-cache*.

The organization of this OS-cache offers several design choices. In our preliminary work, a small subset of these choices have been explored. We consider such a cache at the L1 and L2 levels. We allow data to be simultaneously cached in both the user and OS caches (*i.e.*, the user and OS caches are not mutually exclusive). As a result, cache coherence is required between the user and OS caches and a miss in one of these caches must trigger a look-up in the other before the request is sent to the next level. When dealing with a load or store instruction, we must determine which cache to look up first. For our initial design, every load/store instruction issued by the user thread first looks up the user cache. Every time we enter privileged mode, we look up a simple hardware predictor that estimates the run-length of the currently invoked OS syscall. Loads and stores are first steered towards the OS cache only if the syscall is estimated to be moderately long (short syscalls and very long syscalls are most likely to access user data and cause cache coherence misses). Clearly, a richer design space needs to be explored before settling on an optimal OS-cache organization: the implementation of mutually exclusive OS and user caches, policies for which cache is looked up first, policies to avoid looking up both caches before sending the request to the next level, the trade-offs in looking up the caches in series or in parallel, etc.

Our results show that the implementation of separate user and OS caches at the L1 level offers little benefit. When implementing an additional separate 1 MB OS-cache at the L2 level, a performance improvement of 27-34% is observed for Apache and SPECjbb. This benefit can grow to as much as 60-75% if we assume zero cache coherence overheads, indicating that there is significant room for improvement via intelligent caching policies. Instruction references for user and operating system code are mutually exclusive, providing good cache separation at the L2 level. We also observed that OS syscall run-length is a reasonable indicator of upcoming OS-generated coherence traffic and can be used to help reduce coherence traffic between the user and OS caches. While OS cache separation increases performance, it comes at the cost of additional shared L2 which typically increases performance of all applications. We observe that for OS intensive workloads, a separate OS/User organization can outperform traditional shared L2 utilizing the same total capacity; policies that enable cache partitioning only when performance improvement is expected and default back to a shared L2 of full capacity are quite compelling.

Such an organization alleviates the primary problems with the off-load approach: the high cost of execution migration, the high cost of inter-core cache coherence, and the lack of scalability. A few benefits of off-loading are being compromised here. Firstly, it has been argued that the OS core can be customized to execute OS codes more efficiently. To date, no one has proposed microarchitectural features that can boost the ILP of an OS core, so it is not clear if this argument will hold up. While the OS core can be customized to consume less power, it is possible that some of these techniques can also be applied in the context of our proposed architecture. Similar to the proposals by Li and John (Li and John 2006), a core in our proposed model can employ dynamic frequency scaling or dynamic issue-width scaling when it is predicted to enter a relatively long syscall. Secondly, off-loading syscalls to a single OS core helps consolidate the OS working set to a single cache, enabling high re-use of data and high hit rates for a modest transistor budget. More analysis is required to determine if an additional OS-cache per core is indeed the best use of transistor budgets. It is worth noting that a 1 MB cache can consume as little as 0.75 W of power, while even a modest EV4-like core can consume 7 W (Rusu et al. 2007; Mogul et al. 2008). Therefore, the addition of an OS-cache is perhaps more appealing than the addition of an OS-core.

More analysis is required to justify our approach, but our initial analysis reveals a compelling design point. While it may be appealing to take advantage of a heterogeneous many-core processor to provide customized OS execution, we believe that greater overall efficiency can be achieved by simply adding an OS cache structure or dynamic cache partitioning to each core.

References

Anant Agarwal, John Hennessy, and Mark Horowitz. Cache Performance of Operating System and Multiprogramming Workloads. *ACM Trans. Comput. Syst.*, 6(4):393–431, 1988. ISSN 0734-2071.

Jeffery A. Brown and Dean M. Tullsen. The Shared-Thread Multiprocessor. In *Proceediings of ICS*, pages 73–82, 2008.

Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-Fly. In *Proceedings of ASPLOS-XII*, pages 283–292, New York, NY, USA, 2006. ACM. ISBN 1-59593-451-0.

J. Bradley Chen and Brian N. Bershad. The Impact of Operating System Structure on Memory System Performance. In *Symposium on Operating Systems Principles*, pages 120–133, 1993.

Tao Li and Lizy Kurian John. Operating System Power Minimization through Run-time Processor Resource Adaptation. *IEEE Microprocessors and Microsystems*, 30:189–198, June 2006.

Jeffrey C. Mogul, Jayaram Mudigonda, Nathan Binkert, Parthasarathy Ranganathan, and Vanish Talwar. Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems. *Micro, IEEE*, 28(3):26–41, May-June 2008. ISSN 0272-1732.

David Nellans, Rajeev Balasubramonian, and Erik Brunvand. A Case for Increased Operating System Support in Chip Multi-processors. In *IBM Annual Thomas J. Watson P=ac^2 Conference*, Yorktown Heights, NY, September 2005.

Joshua Redstone, Susan J. Eggers, and Henry M. Levy. An Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture. In *Proceedings of ASPLOS-IX*, pages 245–256, 2000.

S. Rusu, S. Tam, H. Muljono, D. Ayers, J. Chang, B. Cherkauer, J. Stinson, J. Benoit, R. Varada, J. Leung, R. Lim, and S. Vora. A 65-nm Dual-Core Multithreaded Xeon Processor With 16-MB L3 Cache. *IEEE Journal of Solid State Circuits*, 42(1):17–25, January 2007.

Richard Strong, Jayaram Mudigonda, Jeffrey C. Mogul, Nathan Binkert, and Dean Tullsen. Fast Switching of Threads Between Cores. *Operating System Review - To Appear*, April 2009.