Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-fly

Koushik Chakraborty Philip M. Wells Gurindar S. Sohi

Computer Sciences Department
University of Wisconsin, Madison
{kchak, pwells, sohi}@cs.wisc.edu

Abstract

In canonical parallel processing, the operating system (OS) assigns a processing core to a single thread from a multithreaded server application. Since different threads from the same application often carry out similar computation, albeit at different times, we observe extensive code reuse among different processors, causing redundancy (e.g., in our server workloads, 45–65% of all instruction blocks are accessed by all processors). Moreover, largely independent fragments of computation compete for the same private resources causing destructive interference. Together, this redundancy and interference lead to poor utilization of private microarchitecture resources such as caches and branch predictors.

We present *Computation Spreading (CSP)*, which employs hardware migration to distribute a thread's dissimilar fragments of computation across the multiple processing cores of a chip multiprocessor (CMP), while grouping similar computation fragments from different threads together. This paper focuses on a specific example of CSP for OS intensive server applications: separating application level (user) computation from the OS calls it makes.

When performing CSP, each core becomes temporally specialized to execute certain computation fragments, and the same core is repeatedly used for such fragments. We examine two specific thread assignment policies for CSP, and show that these policies, across four server workloads, are able to reduce instruction misses in private L2 caches by 27–58%, private L2 load misses by 0–19%, and branch mispredictions by 9–25%.

Categories and Subject Descriptors C.1.2 [*Processor Architectures*]: Multiple Data Stream Architectures (Multiprocessors); B.3.2 [*Memory Structures*]: Design Styles— Shared Memory

General Terms Design, Performance, Experimentation

Keywords Cache Locality, Dynamic Specialization

1. Introduction

In the canonical model of assigning computation from multiple threads to multiple processors, an entire software thread — including any operating system calls it makes — is assigned to a single processor for execution. This was, perhaps, the only practical approach for traditional multiprocessors built from multiple chips. But since there is commonality among the computation performed by the different threads (i.e., commonality in the code executed), this distribution leads to inefficient use of the microarchitectural structures of the individual processing cores, such as private instruction caches and branch predictors.

With emerging technology trends for chip-multiprocessors (CMPs), ample opportunities exist for alternate solutions. These trends include applications that exhibit large code and data footprints and make extensive use of the operating system (OS) [8, 23], and upcoming support for hardware migration of processor state, similar to Intel's Virtualization Technology (VT) [30]. In the light of these trends, this paper proposes *Computation Spreading (CSP)*, a new model for distributing different fragments of a thread's computation across multiple processing cores in a CMP using hardware thread migration. We define a *computation fragment* as an arbitrary portion of a dynamic instruction stream. Conceptually, CSP aims to collocate similar computation fragments from different threads on the same core while distributing the dissimilar computation fragments from the same thread across multiple cores. Each CMP core thus becomes dynamically and temporally specialized for executing a set of specific computation fragments by retaining the states (such as instruction cache contents and branch predictor entries) necessary to perform each computation efficiently.

As a specific application of this model, we present two assignment policies which separate the execution of system calls and interrupt handlers from the execution of user code, and distribute these two dissimilar computation fragments to different CMP cores. *Thread Assignment Policy* (TAP) prefers to run the OS (or user) portion of a thread on the same core repeatedly, aiming to reduce OS and user interference while maintaining data and instruction locality for each software thread; *Syscall Assignment Policy* (SAP) prefers to run a particular system call (e.g., *read()*) on the same core repeatedly, regardless of which thread made the call, aiming to further improve instruction locality and take advantage of any data structures shared among multiple dynamic instances of the same system call. Both provision a subset of the processing cores for executing user code, and the remainder for the OS.

Unlike previous research on separating OS and user execution, which primarily considered one or more single-core processors [2, 18, 24, 25, 28], TAP and SAP are able to alleviate the interference of separating dissimilar tasks *and* benefit from the symbiosis of collocating similar tasks.

In this paper, we make several contributions:

- We examine the code reuse characteristics of four multithreaded server workloads, plus a parallel make benchmark (pmake), and explore the synergy of locating similar computation fragments on the same cores. Our results show that most instruction blocks are accessed by many, if not all, CMP cores, implying that they

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.

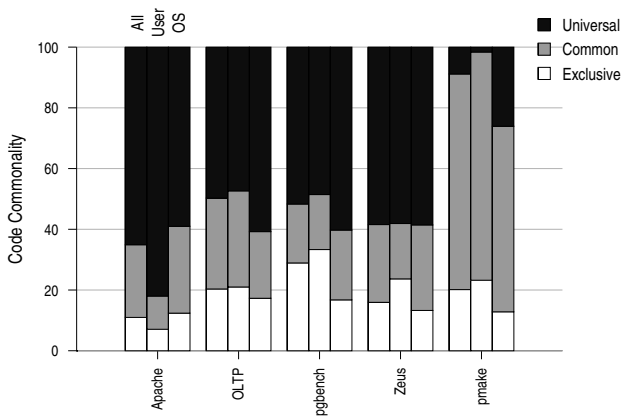


Figure 1. Code Commonality Characteristics for an 8-core CMP. *Each 64-byte static instruction block is marked* Universal *if accessed by all the cores,* Common *if accessed by more than one core, and* Exclusive *if accessed by only one. The left bar shows the commonality profile for the entire execution, the middle bar shows commonality for only user code, and the right bar shows the commonality for OS code.*

all execute similar computation fragments (albeit at different times), and the canonical model of work distribution leads to inefficient use of the aggregate cache space. (Section 2)

- We propose *Computation Spreading (CSP)*, which distributes a thread's execution to multiple cores based on the similarity of individual fragments of the computation performed by the thread, and examine two specific assignment policies (TAP and SAP) for spreading user and OS execution across different CMP cores. (Sections 3 and 5)
- For the four server workloads, we demonstrate that TAP and SAP can reduce L2 instruction misses by 27–58%, while L2 load misses see a 0–19% decrease. Branch mispredictions are reduced by 9–25%, and on-chip interconnect bandwidth is reduced by 0–12%. Overall, these policies result in a performance improvement of 1–20%. The results for pmake, a fifth, non-server benchmark, are not as favorable. (Section 6)

2. Multiprocessor Code Reuse

In the canonical software architecture for multithreaded server applications, a software thread is launched to process an input data item or request. Multiple threads are used to process multiple requests. A thread is scheduled for execution on a single processing core, and all the phases of execution for that thread are typically run on that core (in the absence of OS-induced thread migration). In a multiprocessor, concurrent threads execute on different cores, if possible. Since these separate threads are likely to act on most requests in a similar manner, they traverse through similar code paths (possibly with different data) and end up executing the same instruction blocks.

To quantify this code commonality, we profile instruction accesses for five multi-threaded workloads on a simulated 8-core CMP, and examine how many processing cores fetch each (static) instruction cache block.[1] Figure 1 shows this commonality for the overall execution (left bar), application-level, user code (center) and operating system code (right). Each bar shows the fraction of 64-byte instruction blocks fetched by a core that are fetched by all the processing cores (*Universal*), by more than one core (*Common*), and by only one core (*Exclusive*).

[1] The workloads and our target CMP system are described in more detail in Section 4. The length of these runs is shown in Table 4.

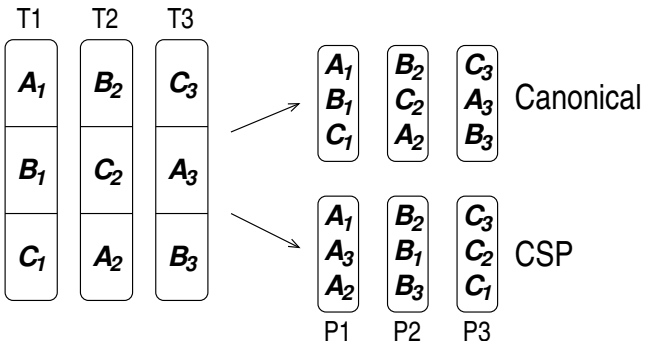


Figure 2. Computation Spreading vs. the Canonical Model for mapping computation fragments from a thread. A_1, A_2 *and* A_3 *denote similar fragments from different threads;* A_1, B_1 *and* C_1 *denote dissimilar fragments of the same thread.*

Several observations can be made from the figure. First, there is a significant amount of commonality in the code executed on the different cores, both for user code and system code. Second, there is little code that is exclusively executed on a single processing core. Third, we notice that the extent of universal sharing is more pronounced for OS code than for user code in three benchmarks (OLTP, pgbench, and pmake) while the opposite is true for Apache. This extensive sharing of instruction blocks in database workloads running on simultaneous multi-threaded (SMT) processors was also observed by Lo, et al. [20], and for other server workloads by Beckmann and Wood [9].

Commonality in the code executed by the different processing cores results in the same instruction blocks being referenced by different cores. This leads to two distinct outcomes: (i) *reference spreading* [19], which refers to the multiple references to a single cache block across multiple processors, leading to a cache miss in each of those caches, and (ii) the replication of the code in multiple caches and subsequent reduction in aggregate cache space. This combined effect can drastically degrade the hit rate, especially for applications with large instruction footprints.

In this paper, we propose using hardware migration to explore different ways of distributing a thread's computation across CMP cores, and aim to provide a more cooperative framework to efficiently utilize on-chip resources.

3. CSP: An Overview

The idea behind *Computation Spreading* (CSP) is to spread out the dissimilar, and perhaps independent, computation fragments of a single thread across multiple processing cores, and concentrate similar fragments from multiple threads onto a single processing core. A depiction of CSP is illustrated in Figure 2. Consider three threads of execution: $T1$, $T2$ and $T3$. Each thread goes through three specific fragments of computation during a given period of time. In the canonical model for mapping computations, the entire execution of a thread is mapped onto one processing core (for example, $T1$ is mapped onto $P1$). In CSP however, the execution is mapped according to the specific computation performed. In this particular case, the similar computation fragments A_1, A_2 and A_3 are all mapped to $P1$, and when any of the threads execute these similar fragments, they are always executed on $P1$. Likewise, the other computation fragments are mapped to $P2$ and $P3$. Thus, dissimilar computation fragments (e.g., A_1 and B_1) are assigned to different processing cores, allowing each core to specialize its

microarchitectural resources for a particular type of computation fragment.

In general, a well-written, modular program is composed of a set of fairly disjoint fragments (e.g., functions in high-level language). Two fragments can be considered similar when they carry out a related set of tasks, and thus traverse closely related code paths and/or have frequent data communication. Likewise, two fragments performing unrelated tasks, with little common code, will be inferred as dissimilar (e.g., page fault handling and process scheduling).

There are two key conceptual issues for CSP: *dynamic specialization* and *data locality*. By localizing specific computation fragments on a single core, CSP allows that core to retain many predictive states necessary for the efficient execution of those fragments (like instruction cache contents and branch predictor states). Collectively, different CMP cores, with the same microarchitecture, become dynamically and temporally specialized for different computations, leading to more efficient overall execution.

Since CSP attempts to exploit code affinity as opposed to data affinity[29], data locality can potentially suffer. However, different parts of a large computation do not communicate arbitrarily. Thus, it is possible to retain, and even enhance, the locality of data references through careful selection of the computation fragments.

In this paper, we present a specific example of CSP where we exploit the natural separation between user code and privileged, OS code, and assign separate processing cores for carrying out these distinct fragments of computation. It may be possible to identify other, more general, fragments in various ways, such as profile driven binary annotation or high-level directives from the application itself. However, such general classification of computation fragments is beyond the scope of this paper.

3.1 Technology Trends Enabling CSP

Modern server workloads, such as back-end databases and web servers, make extensive use of the OS, and exhibit large instruction and data working sets, which leads to many misses in the branch predictors and caches that are in close proximity to the processing cores [8, 23]. However, the very nature of these applications inspire CSP and its ability to collocate computation fragments that can utilize the predictive state already present in the caches and branch predictors of a particular core.

While software trends make CSP promising, hardware trends can make CSP feasible. Particularly important is the abundance of cores on a single chip, and the orders of magnitude reduction in communication latency between these on-chip cores as compared to processors on separate chips. Together these trends allow CSP to move threads among CMP cores at a much higher frequency than a traditional OS would. Minimal additional hardware is required for CSP, since much of the necessary support for these hardware migrations already exists in the new generation chips from Intel and AMD with their VT and SVM technologies, respectively [30, 1]. We discuss the specific mechanism we use for thread migration in Section 5.2.1.

4. Evaluation Methodology

For this study, we use Simics [21], an execution driven, full-system simulator which models a *SunFire 6800* server in sufficient detail to boot unmodified OSs and run unmodified commercial workloads. In this section we further describe our simulation environment, workloads, and evaluated CMP architecture.

4.1 Simulation Infrastructure

We have augmented Simics with a detailed, execution driven, out-of-order (OOO) processor and memory timing model using the Micro Architectural Interface (MAI). Wrong-path events, including speculative exceptions, are faithfully modeled. Simics functionally models UltraSparc IIICu CPUs, which implement the SPARC V9 ISA, and we use our timing model on top of it to enforce the timing characteristics of a dual-issue, OOO processor (see Table 2).

Apache	We use the Surge client [7] to drive the open-source Apache web server, version 2.0.48. We do not use any think time in the Surge client to reduce OS idle time. Both client and server are running on the same, 8-processor machine.
OLTP	OLTP uses the IBM DB2 database to run queries from TPC-C. The database is scaled down from TPC-C specification to about 800MB and runs 192 concurrent user threads with no think time.
pgbench	Pgbench runs TPC-B-like queries on the open source PostgreSQL 8.1beta4 database. The database is scaled to allow 128 concurrent threads.
Zeus	We use the Surge client again to drive the Zeus web server, configured similarly to Apache.
pmake	Parallel compile of PostgreSQL using GNU make with the `-j 64` flag using the Sun Forte Developer 7 C compiler. Runs do not include the any serial execution phases. Unlike the server workloads, pmake consists of multiple processes running in separate virtual address spaces.

Table 1. Workloads used for this study

Fetch, issue, commit	2 instructions / cycle
Integer pipeline	8 stages
I-Window & ROB	128 entries, OOO issue
Load and store queues	64 entries each, w/ bypassing
Store buffer	32 entries, processor consistency
YAGS branch predictor	4k choice, 1k except, 6 tag bits
Private L1 instr. cache	32kB, 2-way, 2-cycle, coherent
Private L1 data cache	32kB, 2-way, 2-cycle, write-back
Private L2 unified cache	256kB, 4-way assoc, 10 cycle load-to-use, 4 banks, 4-stage pipelined, inclusive
On-chip shared L3 cache	8MB, 16-way, 75-cycle load to use, 8 banks, pipelined, exclusive
On-chip interconnect	Point-to-point links, avg 25-cycle latency
3-hop cache-to-cache	105 cycle load-to-use
Main Memory	255 cycle load-to-use, 40GB/sec

Table 2. Baseline processor parameters

The workloads we examine, which are running on Solaris 9, are shown in Table 1. Each of these multithreaded applications expose eight running threads to the hardware. In the baseline system, each of these threads is mapped to a single core in our target eight core CMP, similar to a typical, current generation CMP.

The SPARC architecture traps to the OS to handle all TLB misses. This overhead is extremely large for our workloads (10–30% of execution time), dwarfing other factors that traditionally limit performance. Since most other modern architectures fill TLBs in hardware, we did not want to bias our results (positively or negatively) by incremental changes in TLB performance. Thus, in all our simulations, we model an "infinite" sized TLB, which records all active translations that have previously observed a miss.

4.2 Target System

We model an eight-core CMP, whose relevant configuration parameters are shown in Table 2. We choose to model a CMP with medium-sized, private L2 caches because we believe that private caches, in close proximity to a core, provide a favorable trade-off

	Percent of Instr.		Instr. Before Switch	
Benchmark	**User**	**OS**	**User**	**OS**
Apache	39%	61%	2.3k	3.6k
OLTP	84%	16%	9.3k	1.8k
pgbench	88%	12%	22.1k	2.9k
Zeus	26%	74%	1.7k	5.0k
pmake	83%	17%	24.2k	4.8k

Table 3. Breakdown of user and OS instructions, and the average number of instructions executed before switching from one mode to the other (excluding short TLB and register-window traps).

between aggregate on-chip capacity and latency. We also model an 8MByte, strictly exclusive, shared L3 cache.

The L2 caches maintain coherence using an invalidate-based, MOSI directory protocol. The directory maintains shadow tags of the private L2s, and is located at the appropriate L3 bank. We use a point-to-point, crossbar-like interconnect which maintains FIFO order among source-destination pairs.

4.3 Experimental Methodology

Due to inherent variability (primarily from interrupt processing and OS scheduling decisions) as described by Alameldeen, et al. [4], we add a small random variation to the main memory latency, and run several trials of each benchmark per experiment. We generally present average results, and include the 95% confidence interval on performance graphs.

All workloads are run using functional simulation to warm up the OS, applications and infinite TLB, then run for a shorter amount of time to warm up the L3 cache before we begin timing simulations. Private L1 and L2 caches are *not* warmed up because we cannot do so in a way that is beneficial to both the baseline and our proposed schemes. Instead, we run timing simulations long enough to incur 30–130 times as many L2 misses as there are L2 cache lines.

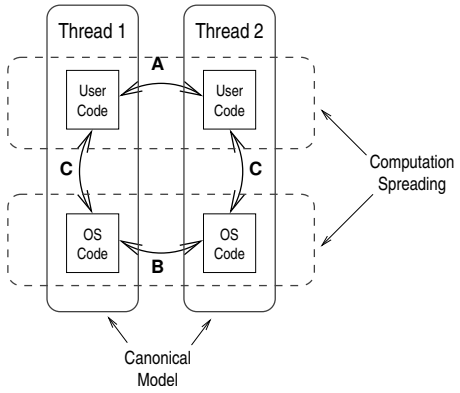
For Apache and Zeus, which have short, roughly uniform length transactions, we run each trial for 1300 transactions. The other benchmarks have long, asymmetric transactions, thus we run each trial for 100M committed *user* instructions to reduce variability. Our results, and other's [33], demonstrate that for these types of workloads, committed user instruction are a good proxy for work-related metrics.

5. An Application of CSP: Separating User and OS Execution

5.1 Why Target User and OS?

Targeting CSP at the user/OS code boundary is appealing for several reasons. First, many commercial workloads spend a considerable amount of time executing both user and OS code, and frequently switch between the two (Table 3). Second, there is a clearly defined separation between these two modes of computation that is easy to identify and exploit with minimal profiling requirements. Third, this clear separation leads to mutually exclusive instruction cache contents and branch predictor states of user and OS code. By collocating these OS and user fragments of computation, however, the canonical model causes extensive destructive interference in these two micro-architectural structures as observed by us and others [2, 18]. And finally, though seemingly counter-intuitive, OS and user fragments have limited data communications between each other, and therefore have a reasonably independent data footprints.

Figure 3(a) illustrates the nature of data communication between OS and user computation in the CMP system we model. Consider two threads running on two different cores in the canonical model. Each thread performs both user and OS fragments of computation; in all there are four separate types of fragments



(a) *Comparing Models*

Benchmark	A	B	C	% of Refs.
apache	1.78	5.58	1.00	4.89%
oltp	7.48	4.40	1.00	3.95%
pgbench	3.03	8.62	1.00	2.22%
pmake	4.42	1.04	1.00	1.69%
zeus	0.81	12.8	1.00	7.43%

(b) *Relative Communication*

Figure 3. Communication Profile between OS and User. *Labels, in both the figures, denote different types of communication: between user computation fragments of two threads (A), OS fragments of two threads (B) and between user and OS computation fragments (C), both in the same and different threads. We show the relative communication between these three groups in (b).*

(two from each thread). The canonical model groups the user and OS computation fragments of a given thread together on one core (shown by the solid rectangles), whereas CSP groups user fragments together, separate from OS fragments (shown by the dotted rectangles).

The arcs represent communication between the various computation fragments. Arc A, user-to-user communication, and arc B, OS-to-OS communication, are both a result of directly accessing locks and data structures that are shared among threads. OS-induced thread migration also contributes slightly to A and B. Arc C is a result of the OS reading and writing data structures passed as pointers to or from a system call.

Figure 3(b) presents the amount of communication between various computation fragments, relative to the communication of user code with OS code (the arcs labeled C). We define communication in this example as any data access (load or store) to a cache line whose previous access was from a different type of computation fragment. For comparison, the last column shows the percent of *all* data references represented by the sum of these three types of communication. While this fraction is small, note that communication between fragments on different cores *always* leads to a cache miss, and is a major contributor to overall memory latency.

In every case except Zeus, there is more communication between user code on different threads than there is between user and OS code (regardless of which thread). Similarly, in every case, we see more communication between the OS code of different threads than between user and OS code.

For an I/O intensive, data streaming application such as Apache, the fact that there is little communication between user and OS code may seem counter-intuitive. But consider that Apache predominantly uses the *sendfile()* system call to tell the OS to copy data from one file descriptor to another: the user code simply di-

rects the data copying, but the OS performs the copy entirely within the OS computation fragment.

When choosing a boundary for spreading execution among cores (which we must do in any model to achieve parallelism), it appears logical, given the data in Figure 3(b), to divide user and OS execution among cores rather than keep user code or OS code from the same thread on a particular core and arbitrarily distribute threads among cores.

5.2 Hardware Support for CSP

Much of the required support for CSP is already provided by the new generation of chips from Intel and AMD using their VT and SVM technologies, respectively [30, 1]. With the increasing popularity of virtualization, it is likely that other vendors will also offer support for hardware thread migration. As the microarchitecture details of these technologies are not available, below we describe the hardware support we assume for CSP, so that the computation from a given thread can move between different CMP cores transparently to the OS software.

5.2.1 Migrating Computation

To move the computation of a thread from one CMP core to another, we need to move its state. We consider two specific types of state a thread may carry: *architected state* and *microarchitectural state*. Architected state consists of a thread's memory and register values and must be preserved. A thread's memory state can simply be communicated as needed via the on-chip coherence network already present to support shared memory multiprocessing. Registers, described in more detail below, must be saved and restored, similar to an OS saving the state of a process when it is context-switched. Microarchitectural state, consisting of predictive state such as cached data and branch predictor entries, need not be preserved for correctness.

The UltraSPARC IIICu architecture we model has a large number of architected registers. Including windowed GPRs, alternate global, floating-point, privileged, ASI-mapped, and TLB control registers, this comprises 277 64-bit registers, or 2.2kB (nearly half of which is SPARC register windows). In other architectures without register windows, register values are saved in the memory across function calls, thereby reducing the architected register state.

While it is possible to build a fast switching network to communicate thread state between different CMP cores, we chose a method requiring limited hardware support for a conservative estimate of thread migration cost. Thus, when a thread executing user code makes a system call, the hardware stores the registers one at a time to the cache. Once that is complete, the appropriate OS core loads the respective registers, again in a serial fashion. At the end of the system call, the thread state is communicated back to the user core. We have not optimized the cache coherence protocol in any way for this transfer. Despite this, the runtime overhead of the migration is relatively small (0.9–2.4% of runtime, discussed later in Section 6).

5.2.2 Virtual Machine Monitor

We assume that a thin Virtual Machine Monitor (VMM) is running underneath the OS software stack, which use the hardware mechanism for transferring thread state through the memory subsystem. The VMM sets aside a portion of the physical address space for this register storage, and directs the state migration as required. Interrupts are delivered to the VMM, which then routes them to the appropriate CMP core. Note that, unlike VMMs which support multiple guest OSs, CSP does not require the virtualization of memory, I/O devices, privileged instruction, or additional security measures. Thus, the functions of the VMM (maintaining a mapping of threads to cores and directing thread migration) are very simple, and could be done in the hardware or simple micro-code with very low overhead.

We fully model the cost of migrating computation in our evaluation of CSP, both due to the effects of caching thread state (by competing for space with other data) and the latency of the migration itself. However, we do not model additional latency of the VMM choosing the destination core of a migrating thread (which we expect to be a very small fraction of the total migration cost).

Since we cannot simply pause a running thread in the hardware without incurring excessive overheads in a multi-processor operating system (due to synchronization between different CPUs [31]), we optimistically assume that we have up to four hardware contexts for concurrently executing multiple threads. Therefore, when the VMM decides to move the computation from a thread onto a core which is already executing another thread, we simply allow the new thread to concurrently run on that core (unless all of its hardware contexts are occupied), sharing the caches and branch predictors but maintaining its own reorder buffer. In the rare event, which has minimal effect on OS synchronization overhead, when all the hardware contexts are occupied, the migrating thread is paused until one of them is again available. However, this concurrent execution also leads to additional capacity and bandwidth pressure in the caches that would not exist if the threads were run serially, which negatively impacts our proposed schemes.

5.3 CSP Policies

Targeting CSP at OS and user executions provides us a simple means to determine when the VMM should consider moving a thread (as the thread enters or leaves the OS). User code invokes the OS for several reasons, such as system calls, interrupts or exceptions. SPARC V9 uses software handling of TLB misses and register window fills and spills, and these short-running traps dominate user-to-OS transitions. In this paper, we consider these traps as user computations (unless a page fault occurs) since they are short running and many other architectures handle them in the hardware.

System calls, interrupts, and page faults — the events which trigger thread migration in CSP — are relatively frequent and long-running. Table 3 shows, on average, how many user instructions pass between these events, and how many instructions are spent processing these events. When these events occur, the core notifies the VMM, which then migrates the computation to another CMP core as dictated by the specific assignment policy. At the end of the event handling, the VMM moves the computation back to its original core.

In all of our simulations, we statically partition eight CMP cores between the user and OS code based on their relative execution time and cache behavior. For Apache and Zeus web servers, we provide two user cores and six OS cores. For pgbench and OLTP database workloads and for pmake, we use the opposite: six user cores and two OS cores. We chose this distribution based on the relative time spent in user and OS mode, respectively. However, this distribution can also be determined dynamically after a brief but representative profiling phase, which we leave for future work. We examine two assignment policies in this paper.

Thread Assignment Policy (TAP): The most straight-forward way for splitting user and OS execution from a thread is to maintain a static mapping of OS and user cores for every thread. At any given time, there are eight running threads to map onto two or six cores, as the case may be. For example, Apache is provisioned with two user cores, so four of the eight threads execute their user fragments on one core, and the other four threads execute their user portion on the second user core. The OS portion of the eight threads needs to be spread across six cores. Since we use a mapping which maintains strong thread affinity for the user or OS fragments, at

Bench.	Instr.	L2 I MPKI	L2 Ld MPKI	L2 St MPKI	L3 MPKI
Apache	94M	22	13	10	3.3
OLTP	120M	20	10	6.2	3.1
pgbench	113M	7.2	7.7	1.6	5.2
Zeus	98M	16	14	12	11
pmake	120M	3.9	3.1	1.9	1.5

Table 4. Baseline Misses per Thousand Instructions (MPKI). *Instr. is total instructions for each trial. L2 I, L2 Ld, and L2 St MPKI columns break down instruction, load and store misses in the unified L2. L3 MPKI comprises all off-chip misses. All results are combined user and OS.*

any given time, two OS cores each have the OS fragments of two threads mapped to them, and four cores each have the OS fragments of only one thread. Throughout this paper, we refer to this policy as *Thread Assignment Policy (TAP)*.

To prevent this load imbalance (on the cores) from creating permanent performance asymmetry for some threads, the VMM changes the mapping every few million cycles. This is necessary because, as Balakrishnan, et al., showed [6], the throughput of these workloads can be negatively affected by performance asymmetry of individual threads.

Syscall Assignment Policy (SAP): Though TAP is straightforward, further dynamic core specialization is possible by grouping similar OS computation fragments onto a subset of the OS cores, and spreading dissimilar OS fragments to the other OS cores. The similarity of different OS fragments is simply detected by recognizing the specific system call (based on register `%g1` in Solaris), or interrupt handling routine, being invoked. For this study, we use a static mapping of particular system calls (plus interrupts and page faults) to cores. Our assignment policy allows a particular system call to be mapped to one or more cores, and will only execute that call on those cores. The policy also allows other system calls to be mapped to the same cores. We refer to this policy as *Syscall Assignment Policy (SAP)*.

During startup, we provision the most common OS computation fragments onto different cores based on their L1 data cache miss component in the baseline (this matches well with the time spent in those computation fragments). Other system calls that we detect at runtime are assigned to a single core in a round-robin fashion. This provisioning scheme aims to distribute the resource demands across different OS cores evenly. Again, online profiling can be used to perform dynamic assignment.

6. Results

Table 4 presents the number of instructions and L2 misses per thousand instructions (MPKI) for the entire workload runs on the baseline system. The poor locality of instruction references causes more instruction misses in the private L2 caches than load misses, making instruction locality critical for performance. Pmake has many fewer L2 misses than the four server workloads, and as such, instruction and data locality, especially at the L2, play a less significant role.

6.1 Branch Prediction

By spreading dissimilar computations across multiple CMP cores, CSP avoids destructive interference of multiple instruction streams (OS and user code) in the branch predictor.Table 5 shows the performance of the YAGS[11] predictor under different schemes. With CSP, we observe a large reduction in mispredictions. For Apache, TAP and SAP reduce the misprediction rate by 15% and 25% respectively. Also, we see that SAP significantly outperforms TAP for Apache and Zeus.

	Branch Misprediction Rates (%)				
Benchmark	**Base**	**TAP**	**SAP**	**SEP**	**SEP 2X**
Apache	6.82	5.65 17.14%	5.09 25.33%	7.09 -4.06%	5.38 21.1%
OLTP	7.12	6.37 10.54%	6.3 11.56%	7.47 -4.85%	6.06 14.87%
pgbench	3.09	2.81 9.09%	2.73 11.72%	2.89 6.36%	2.65 14.17%
Zeus	5.54	4.96 10.56%	4.49 18.95%	6.06 -9.38%	4.85 12.4%
pmake	5.19	5.31 -2.22%	5.29 -1.82%	5.79 -11.4%	4.91 5.47%

Table 5. Branch misprediction rates relative to baseline. *The top row for each benchmark is* Misprediction Rate (%)*, the bottom row is improvement. SEP separates the baseline branch predictor into user and OS halves; SEP 2X separates the predictor into user and OS structures that are each the size of the baseline structures.*

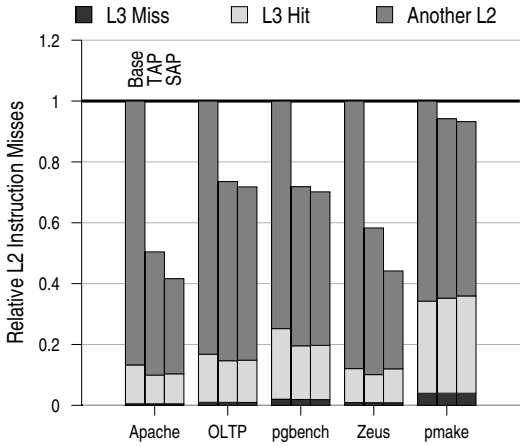


Figure 4. L2 Instruction Misses

Previously, Li, et al., studied the destructive interference in a *gshare* predictor caused by aliases between OS and user entries in the branch predictor, and proposed to mitigate the problem by modifying the branch predictor in each core to use separate tables for user and OS [18]. We model this scheme using a YAGS predictor while retaining the same aggregate predictor space. This is shown in Table 5 as *SEP*. YAGS does not suffer from the alias problem inherent in their baseline gshare predictor, and as such, does not benefit from separating the structures — in fact, except pgbench, it performs worse due to its inability to dynamically adjust the relative number of entries used for user or OS. When separating the predictor for each core into two parts that are each the size of the baseline and twice the aggregate size (shown as *SEP 2X*), significant improvement is achieved. But in nearly every case, SAP performs favorably with SEP 2X despite having half the aggregate size and, more importantly, not requiring any structural changes to the predictor.

6.2 Instruction Cache Performance

Figure 4 shows the normalized L2 misses due to instruction references. The three sets of bars for each benchmark represent the baseline, TAP, and SAP respectively. Each set of bars is broken down into the components of L2 misses: cache-to-cache transfers from another L2, hits in the shared L3, and off-chip misses.

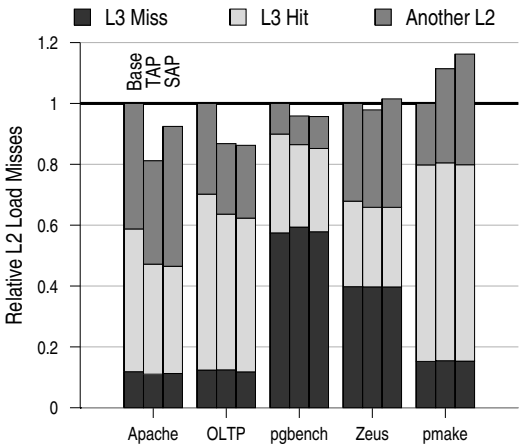


Figure 5. L2 Load Misses

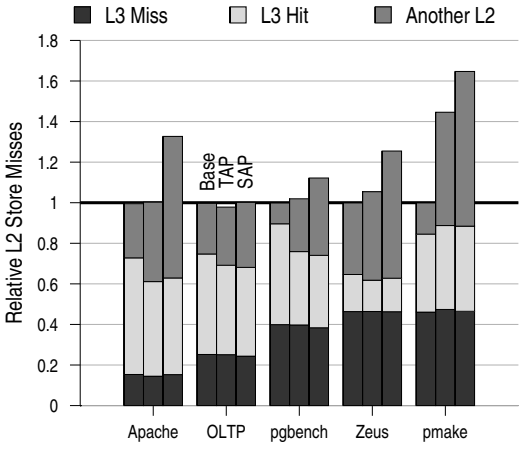


Figure 6. L2 Store Misses

L2 instruction misses are reduced for every benchmark using TAP, since we are spreading the user and OS references (which are completely independent) across different cores and improving locality. This reduction is greater than 25% for every benchmark except pmake. Since SAP assigns the user code in the same way as TAP, it can only affect OS instruction misses. Nonetheless, by running particular system calls on particular cores, SAP further decreases L2 instruction misses for Apache and Zeus.

The two web servers (Apache and Zeus) perform particularly well with both TAP and SAP (up to 58% improvement for SAP), while the performance benefits are more modest among the other applications (27–30% for OLTP and pgbench and 7% for pmake). This result is expected, as the two web servers spend a majority of their time in the OS, and have large instruction miss components from both user and OS. Neither SAP nor TAP improve pmake significantly as it exhibits limited code reuse as evident from Figure 1. L1 instruction misses (not shown for brevity) increase by 7% for pmake, and decrease by up to 20% for Zeus and Apache.

6.3 Data Cache Performance

Similar to instruction misses, we show the L2 load and store misses in Figures 5 and 6, respectively. Again in each figure, the bars are broken down into cache-to-cache transfers from another L2, hits in the shared L3, and off-chip misses. It is important to note that cache misses for lines holding thread state are not included in these figures, though the additional cache misses incurred by normal data lines due to competition with this extra state are included.

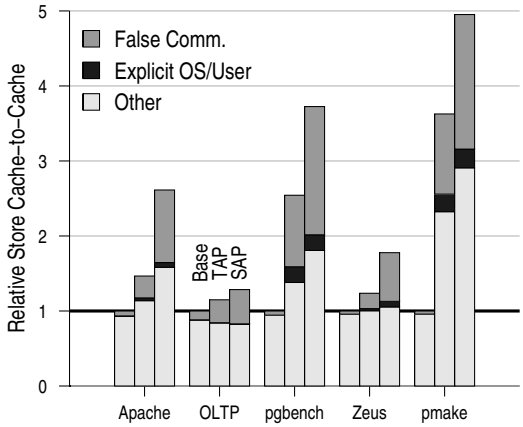


Figure 7. L2 Store Cache-to-Cache Miss Breakdown

For nearly every benchmark, both TAP and SAP slightly increase L1 load misses (20% for pmake and 0–12% for the rest, data not shown), but at the L2 we find the opposite (except pmake): Apache and OLTP, in particular, see 19% and 13% decrease in L2 load misses for TAP, respectively. However, we observe a slight rise in store misses at the L2 for TAP, and a significant rise for SAP, due to an increase in cache-to-cache transfers. OLTP does not have an increase in store misses with SAP because the way we provision its system calls makes SAP very similar to TAP.

To better understand this increase in store cache-to-cache transfers, we classify these misses into three categories in Figure 7: *false communication*, *explicit communication* between OS and user, and *other* cache-to-cache transfers. The total height of the bars in Figure 7 is the same as the upper portion of the bars in Figure 6 (labeled *Another L2*). We identify explicit communication between the OS and user code when the OS reads or writes user data structures with Sparc V9's AS_USER ASIs. As expected, explicit user/OS communication results in negligible misses for the baseline, but even using TAP and SAP this explicit communication is a small fraction of write cache-to-cache transfers.

For TAP, more than 80% of the *false communication* is from the OS making a function call, saving a register window, and incurring a spill trap which spills a *user* register set. Since this spill accesses the user's stack space, which likely resides in a user core's cache, it results in a store cache-to-cache transfer (and later, a load cache-to-cache transfer when the user code incurs a fill trap for this window). Thus, the spill results in needless back-and-forth movement of register window state (since hardware thread migration already moves the register windows to the OS core) leading to additional overheads. The additional false communication for SAP compared to TAP arises from accesses to the OS execution stack (including spill traps) that use the same address region for a given thread regardless of which system call is executing (no data is actually shared on the stack among different system calls). It is likely that the false OS stack communication could be avoided by performing stack address renaming (either in hardware, e.g. [17], or by modifying the OS). For all benchmarks, we find that this false communication is a significant factor contributing to the increase in store misses.

The *Other* category represents communication for which we cannot make a determination. Clearly, much of it is likely to be inherent program communication which might be local in the canonical model but not in TAP and SAP.

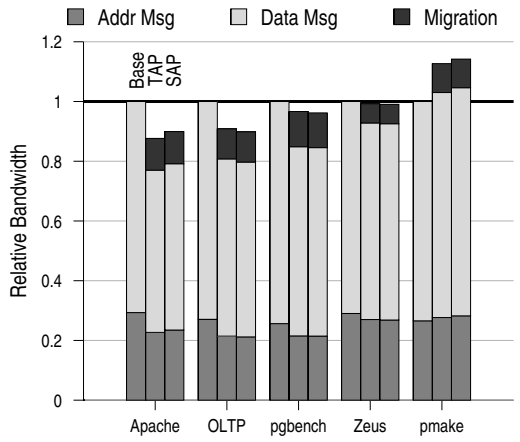


Figure 8. Interconnect Bandwidth Relative to Baseline

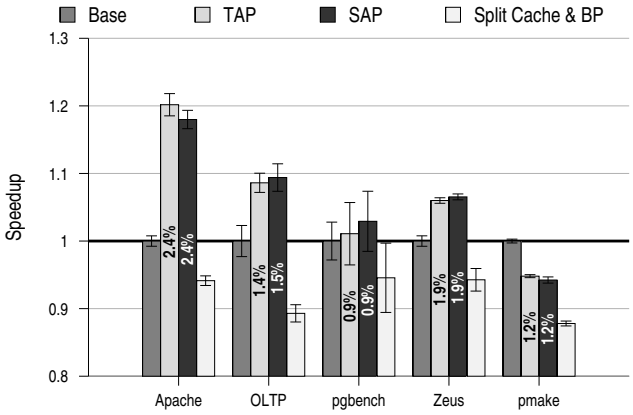


Figure 9. Performance Comparison. *Labels on TAP and SAP bars represent the runtime overhead of thread migration*

Though both pgbench and pmake show a large relative increase in store cache-to-cache transfers, the baseline transfers are much smaller than for the other workloads. Thus, the absolute increase in store cache-to-cache misses is actually quite small.

6.4 On-chip Interconnect Bandwidth

Despite an increase in L2 store misses for most benchmarks, overall L2 misses (which include both instruction and data) decrease for all the server workloads when using both TAP and SAP.

This results in a noticeable improvement in the on-chip interconnect traffic and bandwidth used by these workloads (see Figure 8). This figure breaks down the interconnect bandwidth used into three categories: (a) address messages (requests or responses that do not include data), (b) data messages (data responses and write-backs), and (c) cost of migrating register values between cores for TAP and SAP (see Section 5.2). Except pmake, all other benchmarks see a 0–12% reduction in bandwidth when using TAP or SAP. The bandwidth used for thread migration is 6–12% of the total, despite the large architectural state described in the Section 5.2.

6.5 Performance Impact

Figure 9 shows the speedup using our techniques. CSP yields significant improvements in the average memory response latency and branch misprediction rates for all the benchmarks except pmake, directly impacting the runtime of these workloads. Apache and OLTP see significant speedup, 20% and 8.6% for TAP and 18% and 9.4% for SAP, respectively. Note that the speedups reported here optimistically assume four hardware contexts to execute the different threads as we described in Section 5.2.1.

The labels in the bars for TAP and SAP represent the directly measurable overhead of thread migration, i.e., the number of cycles it takes to store 2.2kB to the local cache and load it from the remote cache. The cost of incurring additional cache misses for other data is modeled, but not included in this label.

As a comparison, the fourth bar of Figure 9 shows the speedup from a configuration where we separate each private cache and branch predictor into separate structures used by the OS and user code. The aggregate cache and predictor sizes at all levels remain the same. As evident from this figure, simply separating the caches and branch predictors to eliminate interference leads to worse performance.

The performance of pmake, our only non-server workload, is much less impressive than the rest. This result is primarily due to two factors. First, pmake shows the least amount of code sharing, especially among user code. Consequently, collocating user computation leads to only marginal improvement in instruction locality (Figure 4). Second, the component of data communication between user and OS is substantially more in pmake than other applications (relative contribution of C in Figure 3(b)). Since OS-to-user communication usually happens within a short interval, it typically leads to local L2 hits in the baseline, while causing cache-to-cache transfers in TAP and SAP. Moreover, especially for pmake, TAP and SAP only save a subset of misses in the other categories, leading to additional data misses overall.

6.6 Code Reuse With TAP/SAP

In Figure 10, we again present the code commonality characteristics for the baseline, this time alongside the two CSP schemes. The *Common* portion from Figure 1 is further broken down into 2 cores and 3-6 cores shown in darkening shades of gray. In every case the *Universal* section virtually disappears for TAP and SAP (as expected). We see a significant increase in the fraction of code used exclusively or by only 2 cores as we move from the baseline to TAP and SAP.

Though we expect a more prominent increase in the codes accessed exclusively or among only 2 cores (as we provision many common system calls to two cores) in SAP, especially for the two web-servers, the results in Figure 10 indicate otherwise. This is mainly because using system call numbers alone is not sufficient for identifying similar and dissimilar computation fragments (or code paths). For example, consider two instances of the *read()* system call. If they are invoked with different types of file descriptors (e.g., a network socket and a regular file), then they are likely to execute substantially different code, though they are executed in the same core in SAP. Likewise, a *read()* and *write()* to the same type of file descriptor is likely to access similar code and data. To fully exploit the potential of CSP, we would need to better distinguish different computation fragments using a more sophisticated technique.

7. Future Work

In this paper, we explore a particular instance of CSP where we distinguish between the user code execution and the OS code execution. This separation is compelling for server applications which spend a substantial amount of time, and incur a substantial number of instruction misses, in both user and OS code. For scientific or other applications which tend to have little OS activity and few instruction misses, neither of the current policies, TAP or SAP, would be advantageous. However, CSP is not limited to separating OS and user computation: even within user code, for example, we can apply hardware thread migration techniques to leverage similar benefits

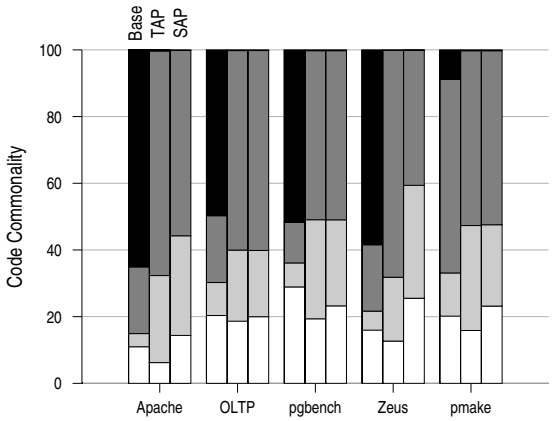


Figure 10. Code Reuse with TAP/SAP. *Each bar is broken down to show different degrees of reuse. From top to bottom these are: Universal, 6–3 cores, 2 cores and Exclusive, respectively.*

for instructions (especially with database workloads). Other class of workloads, such as multimedia or scientific and single-threaded applications, can also benefit from CSP by separating data references between independent computation fragments and improving branch predictions within each fragment.

The current hardware/OS interface limits the completeness of a thorough CSP evaluation. First, as mentioned in Section 5.2.1, if we arbitrarily pause the execution of a thread, the OS will waste a significant number of cycles spinning on synchronization. OS spins can be mitigated using the techniques described by Wells, et al., [31], or eliminated through interface modifications, and we intend to evaluate these ideas in future work. Second, when a thread is paused waiting for the availability of an appropriate core, the thread's former core may be idle waiting for an appropriate thread. This idle time occurs despite the fact that the OS likely has many runnable threads waiting to be scheduled. Exposing all of these runnable threads to the hardware, and letting the hardware choose among them, can potentially alleviate this problem.

8. Related Work

Several recent studies propose to redesign a software application to enforce staged execution, instead of monolithic execution. Such an optimization can significantly improve the performance of server applications, much of which is dominated by memory stall times (with significant contribution from instruction stalls) [3, 13]. One previous work which shares its key motivation with CSP in targeting these stalls is Cohort Scheduling proposed by Larus and Parkes [16]. They argued that the performance problems of server workloads stem from the canonical software architecture of server applications, which results in the execution of non-looping code segments with little instruction and data locality. They propose a new programming model to identify and group related computations and then subsequently schedule them to exploit the similarity between successive computations. SEDA, proposed by Welsh, et. al., decomposes an event-driven server application into multiple stages and schedules them on different processors to enable code reuse and modularity while improving performance and fairness of the system [32]. Harizopoulos and Ailamaki [14] exploit the recurrence of common code paths in transaction processing by identifying regions of code that fit in a L1 I-cache, and frequently context switching among threads to reuse the cache contents. In TAP and SAP, we simply distinguish between user and OS computation and employ hardware thread migration to localize these computations at the hardware level, without specific information about, or modifications to, the software architecture of server applications.

Computation spreading is also quite similar in spirit to LARD [22], which uses locality information at a front-end server to distribute web requests among back-end servers. Load balance is also a major issue with LARD, however it is easier to address because the number of requests is much greater than the number of servers.

Separating OS computation from the user computation is reminiscent of the CDC 6600, which had one high performance Central Processing Unit (CPU), and several lower performance, multithreaded Peripheral Processors (PPs) [27]. The intent was to run threaded, I/O bound, OS activities on the PPs, and save the CPU for heavy computation. When technology eventually allowed an entire processor to fit on one chip, however, the communication costs prohibited executing OS and user code on separate chips.

There have been many studies of OS behavior, and its interaction with the user code on various micro-architectural features [2, 5, 12, 18, 24, 25, 28]. The salient points of most of this research are that 1) many cycles are spent executing operating system code for server applications, 2) OS code has different behavior than user code, and 3) the OS adversely affects many micro-architectural structures through capacity and conflict interference. Our work corroborates most of these claims, and we propose CSP as a potential unified solution to these interference problems.

Several recent proposals aim to improve CMP memory value communication. Most rely on coherence protocol optimizations and/or alternate cache organizations [9, 10, 26]. In this paper, we propose an orthogonal technique to achieve similar goals.

Recent proposals also advocate using heterogeneous CMP cores to achieve performance and power benefits by scheduling the low ILP phases on the slower cores [15]. CSP is a promising match for such hardware, since different computation fragments (such as user and OS) typically have different resource requirements.

9. Conclusions

In this paper, we examine the code reuse characteristics of several multithreaded commercial workloads, and show that most instruction blocks are accessed by many, if not all, CMP cores. This implies that they all go through similar phases of computation (albeit at different times), and that the canonical model of work distribution leads to inefficient use of the aggregate cache space.

We present *Computation Spreading* (CSP), a new model for distributing a thread's computation across multiple cores in a CMP. CSP employs hardware thread migration to spread dissimilar phases of computation from a thread to multiple CMP cores, while localizing similar phases of computation from different threads onto a single core. We apply this technique to separate user computation from the OS computation performed on behalf of the user, and present two assignment policies for this application: TAP and SAP. For four server workloads, both policies achieve a dramatic reduction in L2 instruction cache misses (27% to 58%), while improving L2 load misses, branch misprediction and on-chip interconnect bandwidth for our server workloads. Overall, they lead to reasonable performance improvement of 1% to 20% and 3% to 18%, respectively. Results for pmake, a fifth, non-server benchmark, are not as favorable.

The work in this paper represents an initial foray into a subject that can have significant consequences for the design of chip multiprocessors. While separating user and OS execution is interesting for OS-intensive workloads, Computation Spreading has much potential for further improvement for these and other classes of workloads by more intelligently spreading independent computation within user or OS execution. In this manner, CSP has the potential for completely altering the instruction stream executed on different CMP cores, and will be the subject of much future work.

Acknowledgments

This work was supported in part by National Science Foundation grants CCR-0311572 and EIA-0071924, and by donations from Intel Corporation. We also thank our anonymous reviewers for their comments on this paper.

References

[1] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, Dec 2005.

[2] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst.*, 6(4):393–431, 1988.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, 1999.

[4] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 2003.

[5] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[6] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[7] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the 1998 SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1998.

[8] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th Annual International Symposium on Computer architecture*, 1998.

[9] B. M. Beckmann and D. A. Wood. Managing wire delay in large chip-multiprocessor caches. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, 2004.

[10] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *Proceedings of the 33rd International Symposium on Computer Architecture*, 2006.

[11] A. N. Eden and T. Mudge. The YAGS branch prediction scheme. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, 1998.

[12] N. Gloy, C. Young, J. B. Chen, and M. D. Smith. An analysis of dynamic branch prediction schemes on system workloads. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, 1996.

[13] R. Hankins, T. Diep, M. Annavaram, B. Hirano, H. Eri, H. Nueckel, and J. P. Shen. Scaling and characterizing database workloads: Bridging the gap between research and practice. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, 2003.

[14] S. Harizopoulos and A. Ailamaki. STEPS towards cache-resident transaction processing. In *Proceedings of the 30th International Conference on Very Large Databases*, 2004.

[15] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[16] J. R. Larus and M. Parkes. Using cohort-scheduling to enhance server performance. In *Proceedings of the General Track USENIX Annual Technical Conference*, 2002.

[17] H.-H. S. Lee, M. Smelyanskiy, G. S. Tyson, and C. J. Newburn. Stack value file: Custom microarchitecture for the stack. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001.

[18] T. Li, L. K. John, A. Sivasubramaniam, N. Vijaykrishnan, and J. Rubio. Understanding and improving operating system effects in control flow prediction. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[19] D. Lilja, F. Marcovitz, and P. C. Yew. Memory referencing behavior and a cache performance metric in a shared memory multiprocessor. Technical Report CSRD-836, University of Illinois, Urbana-Champaign, Dec 1988.

[20] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[21] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb 2002.

[22] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[23] A. Ramirez, L. A. Barroso, K. Gharachorloo, R. Cohn, J. Larriba-Pey, P. G. Lowney, and M. Valero. Code layout optimizations for transaction processing workloads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[24] J. A. Redstone, S. J. Eggers, and H. M. Levy. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[25] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[26] E. Speight, H. Shafi, L. Zhang, and R. Rajamony. Adaptive mechanisms and policies for managing cache hierarchies in chip multiprocessors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[27] J. E. Thorton. Parallel operation in the control data 6600. In *Proceedings of the Fall Joint Computer Conference*, 1964.

[28] J. Torrellas, A. Gupta, and J. Hennessy. Characterizing the caching and synchronization performance of a multiprocessor operating system. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[29] J. Torrellas, A. Tucker, and A. Gupta. Benefits of cache-affinity scheduling in shared-memory multiprocessors: a summary. In *Proceedings of the 1993 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1993.

[30] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5), 2005.

[31] P. Wells, K. Chakraborty, and G. Sohi. Hardware support for spin management in overcommitted virtual machines. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, 2006.

[32] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th Symposium on Operating Systems Principles*, 2001.

[33] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.