

Performance of Multi-Process and Multi-Thread Processing on Multi-core SMT Processors

Hiroshi Inoue and Toshio Nakatani

Abstract—Many modern high-performance processors support multiple hardware threads in the form of multiple cores and SMT (Simultaneous Multi-Threading). Hence achieving good performance scalability of programs with respect to the numbers of cores (core scalability) and SMT threads in one core (SMT scalability) is critical. To identify a way to achieve higher performance on the multi-core SMT processors, this paper compares the performance scalability with two parallelization models (using multiple processes and using multiple threads in one process) on two types of hardware parallelism (core scalability and SMT scalability). We tested standard Java benchmarks and a real-world server program written in PHP on two platforms, Sun's UltraSPARC T1 (Niagara) processor and Intel's Xeon (Nehalem) processor. We show that the multi-thread model achieves better SMT scalability compared to the multi-process model by reducing the number of cache misses and DTLB misses. However both models achieve roughly equal core scalability. We show that the multi-thread model generates up to 7.4 times more DTLB misses than the multi-process model when multiple cores are used. To take advantage of the both models, we implemented a memory allocator for a PHP runtime to reduce DTLB misses on multi-core SMT processors. The allocator is aware of the core that is running each software thread and allocates memory blocks from same memory page for each processor core. When using all of the hardware threads on a Niagara, the core-aware allocator reduces the DTLB misses by 46.7% compared to the default allocator, and it improves the performance by 3.0%.

I. INTRODUCTION

Modern high-performance processors provide thread-level parallelism by using multiple hardware threads within each core using Simultaneous Multi-Threading (SMT)¹ [1]. For example, Sun's UltraSPARC T2 (Niagara 2) processor [2] provides 8-way multi-threading in each core, UltraSPARC T1 (Niagara) processor [3] and IBM's POWER7 processor [4] provide 4-way multi-threading, and Intel's Core i7 and Xeon (Nehalem) processors [5] provide 2-way multi-threading. Such SMT processors can increase the efficiency of CPU resource usage by allowing multiple threads to run on a single core, even for workloads with limited instruction-level parallelism. Multiple threads using SMT can also hide memory access latencies. Due to a

growing gap between the processor and memory speed, SMT is becoming more important.

To exploit thread-level parallelism available in such a system, programs have to be parallelized. A programmer can write a parallel program using multiple processes (the *multi-process model*) or multiple threads within one process (the *multi-thread model*) on today's systems. Multiple threads in one process share the virtual memory space, while multiple processes do not share the same memory space. This difference typically results in a larger memory footprint and better inter-process isolation for the multi-process model. For example, the Apache Web server supports both a multi-process model (prefork model) and a multi-thread model (worker model). The prefork model generates multiple Web-server processes to handle many HTTP connections while the worker model generates multiple threads. The prefork model is more reliable but consumes more memory.

In this paper, we study the interactions between the parallelization model and the processor architecture aspects of a multi-core SMT processor. To identify how to achieve higher performance on the multi-core SMT processor, this paper focuses on performance comparisons between a multi-thread model and a multi-process model on two types of hardware parallelism (core scalability and SMT scalability). We measure the throughput of standard Java benchmarks, SPECjbb2005 and SPECjvm2008, using the multi-process model and the multi-thread model, and then we analyze the detailed architecture-level statistics such as cache misses and TLB misses on two processors, Sun's Niagara and Intel's Nehalem. The Niagara has eight cores and four SMT threads in each core, whereas the Nehalem has four cores and two SMT threads in each core. In the measured benchmarks, each thread did not communicate with each other so that we can focus on the architecture-level behavior.

To test the core scalability, we increased the number of cores with only one SMT thread in each core. To test the SMT scalability, we increased the number of SMT threads in each core. Although many previous studies [6-11] measured core scalability and SMT scalability of programs on multi-core SMT processors, this is the first work to focus on the effects of the parallelization model on multi-core SMT processors.

Our results showed that the multi-thread model achieved much better SMT scalability than the multi-process model because the multi-thread model generated a smaller number of cache misses and DTLB misses due to a smaller memory footprint. In contrast to better SMT scalability of the multi-thread model, on average, both models achieved almost

¹A Niagara core cannot execute instructions from multiple software threads in the same processor cycle and thus the multi-threading in Niagara should be called *fine-grained multi-threading*. In this paper, we use the word SMT in a wider sense to include fine-grained multi-threading.

comparable core scalability on Niagara and the multi-process model achieved better core scalability than the multi-thread model on Nehalem. We found that, contrary to our expectations, the multi-thread model generated up to 7.4 times more DTLB misses than multi-process model when we used many cores.

We confirmed that our observation was not specific to Java workloads by testing a real-world server program written in PHP on Niagara. The multi-thread model achieved better SMT scalability than the multi-process model by reducing the DTLB misses, but it showed comparable core scalability.

We also observed that the performance advantage of the multi-thread model was reduced as the number of cores increased. To achieve the better performance on the SMT processors using many cores with SMT threads, we implemented a new memory allocator in the multi-threaded PHP runtime. This allocator avoided sharing a page among threads running on different cores. This reduced the DTLB misses by about half and improved the performance when using all of the hardware threads on Niagara. The modern memory allocators are often aware of the remote and local memories on NUMA systems and control the allocations to minimize the costly remote memory accesses. Our allocator provided similar control at a finer granularity even on a non-NUMA system and improved the performance.

There are two main contributions in this paper. (1) We demonstrate that the smaller footprint of the multi-thread model does not always result in the higher performance and it may degrade the performance compared to the multi-process model due to more frequent DTLB misses on today's multi-core processors. The performance of the multi-thread model and the multi-process model depend on the underlying processor architecture and that the interactions are complicated. (2) We show that a memory allocator that tracks the processor core executing each program can improve the performance on a multi-core SMT processor by significantly reducing the DTLB misses.

The rest of the paper is organized as follows. Section 2 covers related work. Section 3 summarizes the experimental results with Java benchmarks. Section 4 describes our experiments with a larger PHP program. Section 5 describes our core-aware memory allocator and presents performance results. Finally, Section 6 summarizes this paper.

II. RELATED WORK

There are many papers that have measured performance scalability on systems with high thread-level parallelism using multi-core SMT processors [6-11]. For example, Kaseridis and John [7] studied the scalability of the SPECjbb2005 on a Niagara processor. Later, Tseng *et al.* [8] also studied the scalability of Java programs on Niagara. Though these papers focused on the core scalability and the SMT scalability of the benchmarks, they did not study the differences of the multi-process model and the multi-thread model, which is our focus in this paper. Ueda *et al.* [9] studied

the performance scalability of the SAP Java application server on a Niagara processor and on Intel's Xeon (Clovertown) processors, which do not support SMT, while changing the number of JVMs from one to four. The one JVM configuration in their work corresponds to the multi-thread model, and using more JVMs makes the program more like the multi-process model, which is an extreme case of the multiple-JVM configurations. They showed that using more JVMs eased the lock contention while it increased the memory footprint and cache misses. As a result, a 2-JVM configuration achieved the best performance on the Niagara system and a 4-JVM configuration peaked on the Xeon system. They concluded that the amount of cache memory was the key factor to determine the best configuration. Our results showed that the 4-way SMT of Niagara might be another important factor that reduced the performance of 4-JVM configuration on Niagara.

Many previous studies identified lock contention as the dominant cause of poor scalability on multi-core SMT processors [9-11]. For example, Ishizaki *et al.* [11] reported that the performance of Java server workloads running on a Niagara2 processor was improved by up to 132% by removing the contended locks. We found that appropriate use of the parallelization model is also an important factor in improving the performance of programs on multi-core SMT processors. The programs we used in this paper did not actually suffer from lock contention even when we used all of the hardware threads of a Niagara processor. This was because we selected programs that did not share data between the software threads so as to focus on interactions between parallelization model and underlying micro architecture. Server-side programs for Web applications, in which each thread serves a different client request without communicating among each other, are important examples of such programs that did not share data among software threads. Other important examples include Hadoop (an open-source implementation of the MapReduce runtime written in Java) or scientific workloads using MPI.

In this paper, we propose a new memory allocator for multi-threaded programs, an allocator that is aware of the physical core that is running the software threads and allocates memory from the same memory page for each core. The basic idea of such a memory allocator is not new. For example, existing NUMA-aware memory allocators [12] are aware of the physical locations of the software threads and allocate memory blocks from local memory on the NUMA machines. We use a similar technique for a different target.

III. PERFORMANCE RESULTS FOR JAVA WORKLOADS

A. System Setup

To study the performance of programs on a multi-core SMT processor, we choose Sun's UltraSPARC T1 (Niagara) processor, which has 8 cores with 4 SMT threads in each core, and Intel's Xeon X5570 (Nehalem) processor, which has 4

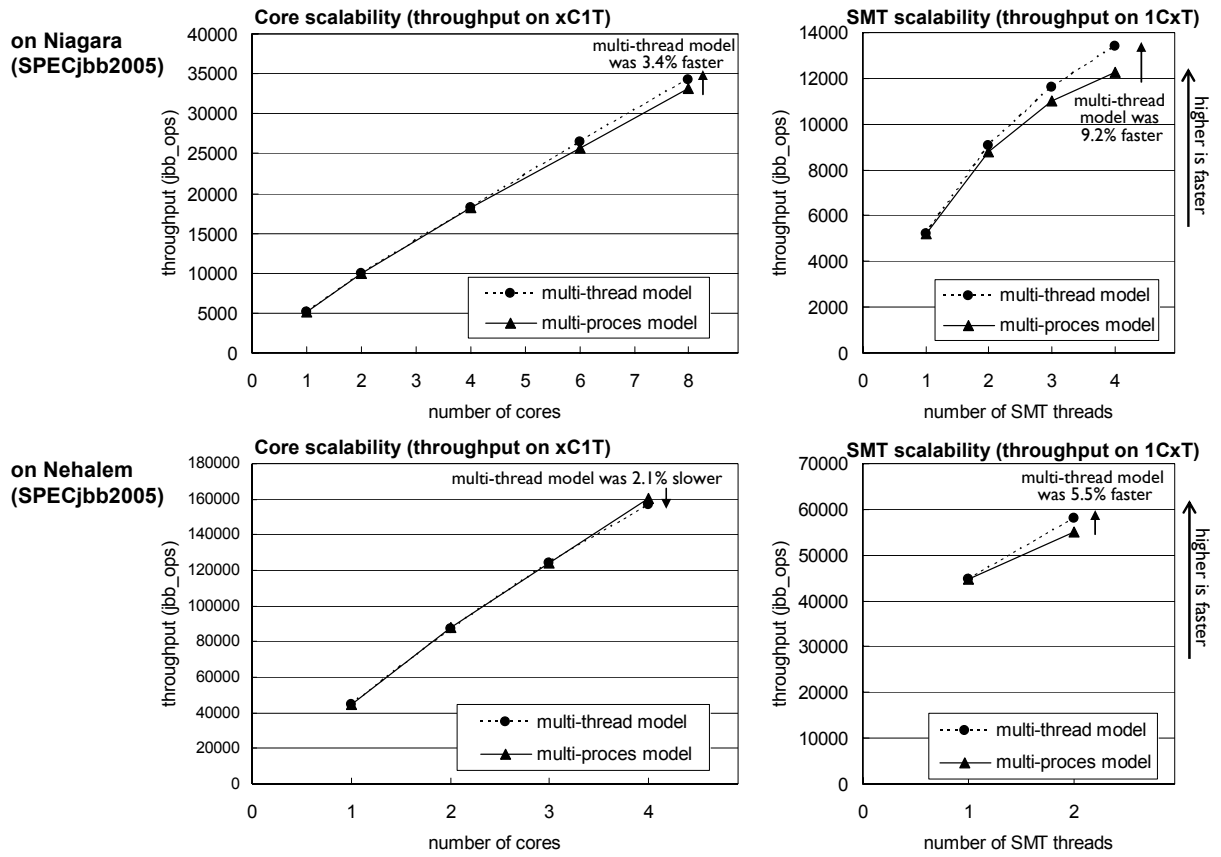


Figure 1. Core scalability and SMT scalability for SPECjbb2005 on Niagara and Nehalem.

cores with 2 SMT threads in each core, as the platform for our evaluations. A Niagara core contains 16 KB of L1 instruction cache, 8 KB of L1 data cache, and 64 entries in its DTLB. The 8 cores all share a 3-MB unified L2 cache. We used a system with a 1.2-GHz Niagara processor and 16 GB of system memory, running Solaris 10. A Nehalem core contains 32 KB of L1 instruction cache, 32 KB of L1 data cache, 256 KB L2 cache, 64 entries for small pages and 32 entries for large pages in a L1 DTLB, and 512 entries for small pages in a unified L2 TLB. The all cores share an 8-MB L3 cache. We used a system with a 2.93-GHz Nehalem processor and 24 GB of system memory, running RedHat Enterprise Linux 5.4.

In this section, we used two standard Java benchmarks, SPECjbb2005 [13] and selected programs from SPECjvm2008 [14]. We picked these benchmarks because they do not share data among the software threads and thus we can run them with the multi-process model and the multi-thread model without having to modify the programs. For example, in SPECjbb2005, which emulates a server side Java workload in a three-tier system, a worker thread accesses only its own Warehouse data set and does not communicate with other threads. Though the default configuration of the SPECjbb2005 is a multi-thread model using one JVM, it also allows the use of multiple JVMs. Thus we can run the SPECjbb2005 as a multi-process model by setting the same value as the number of hardware threads for the number of JVMs. In SPECjvm2008, worker threads do not

communicate with each other and hence we can execute these programs in the multi-process model, though such multiple-JVM configuration is not officially allowed to submit the results. When a program requires temporary disk space, we explicitly specify a different location for each JVM to avoid conflicts among JVMs. We used the 32-bit HotSpot server VM for Java 6 Update 17 on both platforms. We configured the size of the Java heap as 256 MB per software thread. We specified a memory page size of 4 MB (Niagara) or 2 MB (Nehalem) for the Java heap in the JVM command-line options. To control the number of cores and SMT threads used, we explicitly specify the logical processors to use for each JVM by using the tools provided by the operating systems, `psrset` on Solaris and `numactl` on Linux.

B. Core Scalability and SMT scalability

In this section, we describe our experimental results for the core scalability and SMT scalability of the benchmarks with the multi-thread model and the multi-process model. In these results, we use the notation xCyT to show that x cores with y hardware threads per core were used in the experiment [8]. For example, 8C4T means that all hardware threads of a Niagara processor were used in the measurement and 1C1T means only one hardware thread on one core was used.

Figure 1 shows the core scalability and SMT scalability of SPECjbb2005 on two platforms. For the SMT scalability on

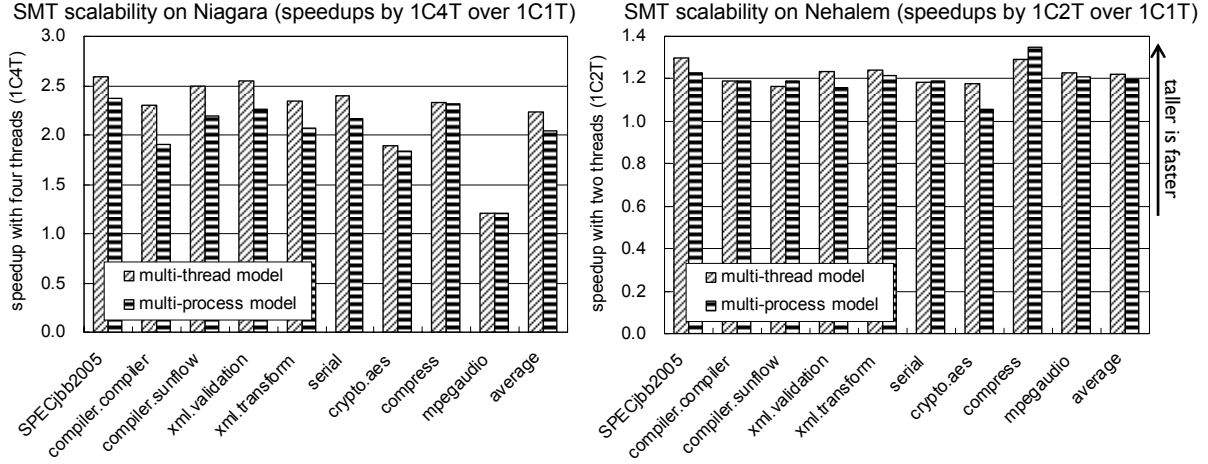


Figure 2. SMT scalability using multiple SMT threads on one core on Niagara (1C4T) and Nehalem (1C2T) for SPECjbb2005 and SPECjvm2008.

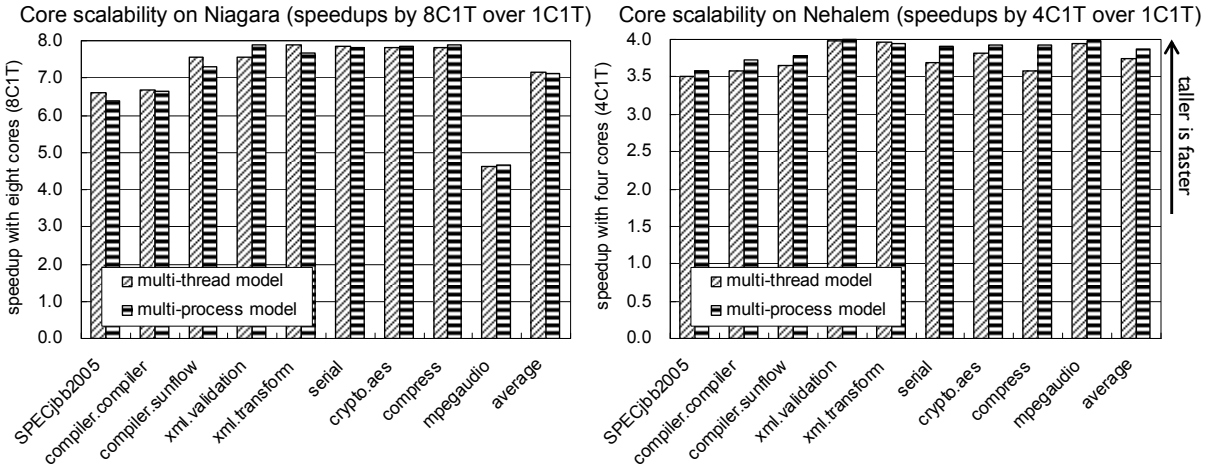


Figure 3. Core scalability using one thread/core on Niagara (8C1T) and Nehalem (4C1T) for SPECjbb2005 and SPECjvm2008.

Niagara, the multi-thread model outperformed the multi-process model when using two or more SMT threads. The maximum performance advantage of the multi-thread model was 9.2% when using four SMT threads (1C4T). For the core scalability on Niagara, the multi-thread model and multi-process model achieved almost comparable speedups for up to four cores. When all eight cores were used, the multi-thread model was 3.4% faster than the multi-process model. We detail the reason in the next section using architecture-level statistics. On Nehalem, the performance difference of the multi-thread model and the multi-process model was smaller. The multi-thread model was faster than the multi-process model using two SMT threads (1C2T) by 5.5% while it was slower using all four cores with one thread each (4C1T) by 2.1%.

To compare the SMT scalability and the core scalability with the multi-thread model and the multi-process model for all workloads, Figure 2 depicts the speedups using all four SMT threads (1C4T) on a Niagara core or all two SMT threads on a Nehalem core (1C2T). On both platforms, the multi-thread model achieved larger speedups using SMT threads (better SMT scalability) than the multi-process model.

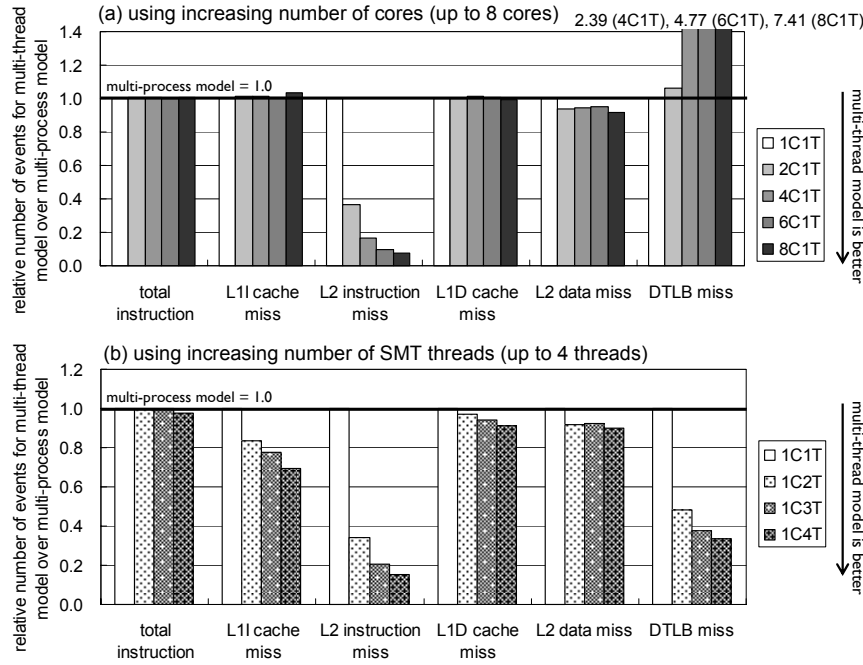
The performance advantages were 9.6% on Niagara and 2.0% on Nehalem on average of the tested programs.

Figure 3 shows the speedups using eight Niagara cores (8C1T) or four Nehalem cores (4C1T) with one thread on each core. Unlike the SMT scalability, the core scalability with the multi-thread model was not better than the core scalability with the multi-process model on both platforms. The performance advantage of the multi-thread model was almost negligible on Niagara and the multi-thread model was 3.0% slower than the multi-process model when using all cores of each processor. In Figure 2 and 3, poor scalabilities in mpegaudio on Niagara were due to the contentions in a floating-point unit because Niagara equipped only one floating-point unit shared by all cores.

C. Architecture-Level Statistics

For further insight into the causes of the differences in the core scalability and the SMT scalability, Figure 4 shows the change in the relative numbers of instructions and cache misses per transaction for the multi-thread model over the multi-process model for SPECjbb2005 on Niagara and Nehalem as we changed the numbers of cores and SMT threads. The bars that are shorter than 1.0 mean that the

**on Niagara
(SPECjbb)**



cache structure of Niagara:

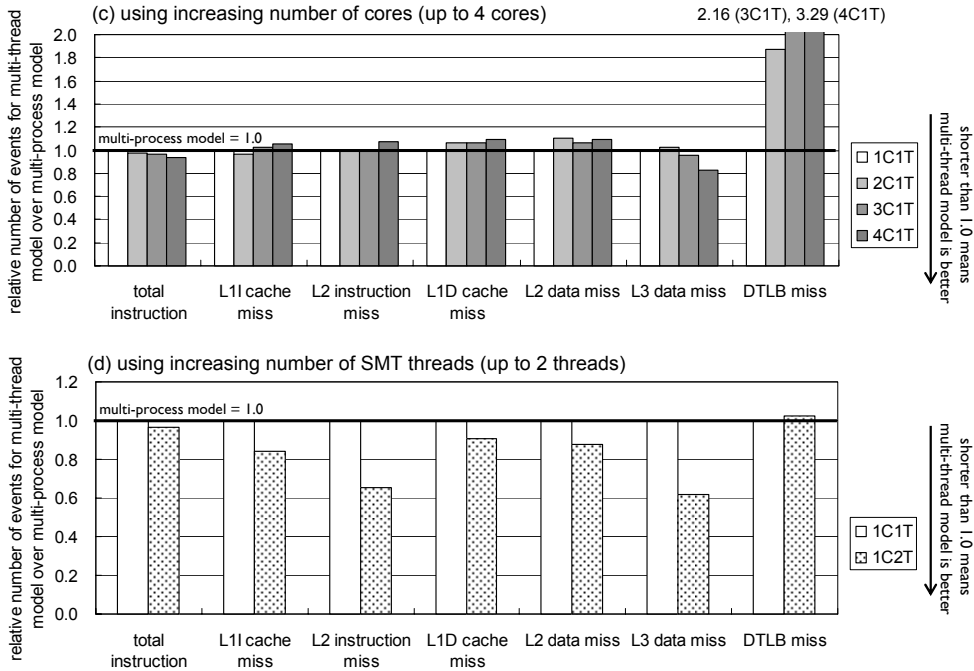
L1I cache:
16 KB
per core
32 bytes/line

L1D cache:
8 KB
per core
16 bytes/line

L2 cache:
3 MB
shared by 8 cores
64 bytes/line

DTLB:
64 entries
per core

**on Nehalem
(SPECjbb)**



cache structure of Nehalem:

L1I cache:
32 KB
per core
64 bytes/line

L1D cache:
32 KB
per core
64 bytes/line

L2 cache:
256 KB
per core
64 bytes/line

L3 cache:
8 MB
shared by 4 cores
64 bytes/line

L1D TLB:
64 entries
(for small pages)
32 entries
(for large pages)
per core

L2 TLB:
512 entries
(for small pages)
per core

Figure 4. Comparisons of the numbers of instructions and cache misses with multi-thread model against multi-process model for SPECjbb2005 with (a) increasing numbers of cores on Niagara, (b) increasing numbers of SMT threads on Niagara, (c) increasing numbers of cores on Nehalem and (d) increasing numbers of SMT threads on Nehalem.

multi-thread model generated fewer cache misses than the multi-process model and hence the multi-thread model was better.

In Figure 4(a), the numbers of instructions and L1 cache misses were not affected by the parallelization model, whereas the numbers of L2 cache data misses and L2 cache instruction misses were smaller for the multi-thread model. These reduced L2 cache misses were due to the smaller footprint of the multi-thread model. In particular, the reductions in the L2 cache instruction misses were significant.

This was because the JVM had a JIT compiler, and thus the instructions generated by the JIT compiler at runtime were not shared among the processes and the instruction footprint increased in proportion to the number of JVMs used.

For the DTLB misses, contrary to our expectations, the multi-thread model generated up to 7.4 times more misses than the multi-process model, though the total memory footprint was smaller for the multi-thread model. This notable increase in DTLB misses was because each worker thread and GC thread needed to access the entire memory space of a

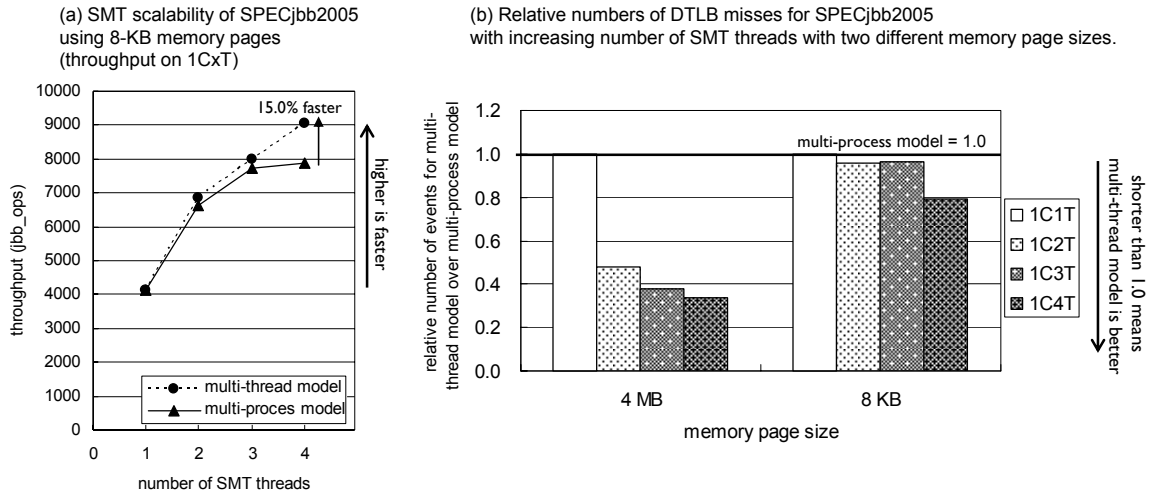


Figure 5. (a) SMT scalability using 8-KB memory pages for SPECjbb2005 on Niagara and (b) relative numbers of DTLB misses with increasing number of SMT threads with two different memory page sizes.

large Java heap (256 MB x the number of threads). Such access patterns resulted in more DTLB misses, because each core needed to preserve the DTLB entries for the entire memory space. In contrast, in the multi-process model each process accessed only its own 256-MB Java heap. A core executing a process needed to preserve only the DTLB entries for a limited memory area and thus the number of DTLB misses was not increased when using more cores. The overall effect of the reduced L2 misses and the increased DTLB misses was that the multi-thread model achieved slightly better core scalability than the multi-process model for SPECjbb2005. From Figure 3, the multi-process model outperformed the multi-thread model in some benchmarks. In these benchmarks, the effect of increased DTLB misses was larger than the reduced cache misses.

On the contrary, as shown in Figure 4(b), the number of DTLB misses for the multi-thread model became smaller than for the multi-process model as we increased the number of SMT threads. Because the SMT threads on one core shared the DTLB, the number of DTLB misses was not increased even when each thread accessed the entire Java heap. The number of L1 and L2 cache misses was smaller for the multi-thread model than for the multi-process model due to the smaller memory footprint of the multi-thread model. As a result, the multi-thread model achieved much better SMT scalability than the multi-process model as shown in Figure 1.

On Nehalem, as shown in Figure 4(c), the number of DTLB misses for the multi-thread model became significantly larger than for the multi-process model when we increased the number of cores. However, the multi-thread model and the multi-process model generated almost comparable number of DTLB misses as shown in Figure 4(d) when we used both SMT threads of a Nehalem core. The multi-thread model reduced L1, L2 and L3 cache misses and these reduced cache misses resulted in a higher SMT scalability of the multi-thread model. Note that L1 and L2 cache are private for each core and L3 cache is shared among

cores on Nehalem, while L2 cache is shared on Niagara.

To confirm that our observations do not specific for HotSpot JVM, we conducted the same evaluations using 64-bit IBM JVM for Java 6, which uses mark-and-sweep GC with flat (non-generational) Java heap while HotSpot VM uses generational GC, on Nehalem. The results with IBM JVM were consistent with the results with HotSpot. For example in SPECjbb2005, the multi-thread model achieved 3.3% better performance than multi-process model using two threads (1C2T), while it was 3.4% slower using four cores (4C1T). We also observed the multi-thread model generated 1.94x more DTLB misses compared to the multi-process model using four cores (4C1T).

To examine how the memory page size affects our observations, Figure 5(a) shows the SMT scalability of SPECjbb2005 on Niagara using 8-KB memory pages instead of 4-MB pages. The multi-thread model achieved higher SMT scalability than the multi-process model even with the smaller pages. The performance advantage of the multi-thread model was increased from 9.2% (with 4-MB pages) to 15.0% (with 8-KB pages).

Figure 5(b) shows the relative number of DTLB misses of the multi-thread model over the multi-process model as we increased the number of SMT threads with the two memory page sizes. We did not include the counts of L1 and L2 cache misses in the figure because they were not significantly affected by the page size. The difference in DTLB misses with the multi-thread model and the multi-process model was smaller for the 8-KB pages than for the 4-MB pages. The multi-thread model generated about 20% fewer DTLB misses than the multi-process model even with 8-KB memory pages. Though this looks smaller than the difference when we used 4-MB pages, the absolute number of total DTLB misses was drastically increased with the smaller page size and hence the difference of 20% had a large impact on performance. For example, the number of DTLB misses per instruction was 0.018% for the multi-thread model with 4-MB pages and

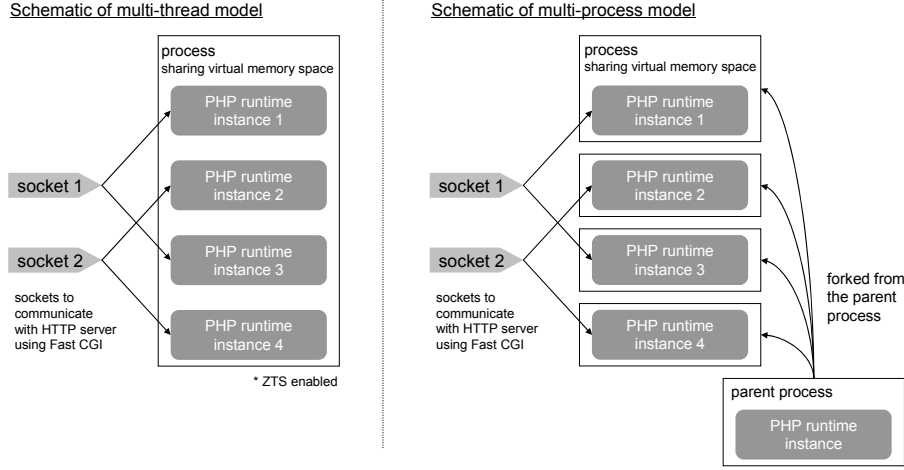


Figure 6. Structures of the multi-thread model and multi-process model for the PHP runtime.

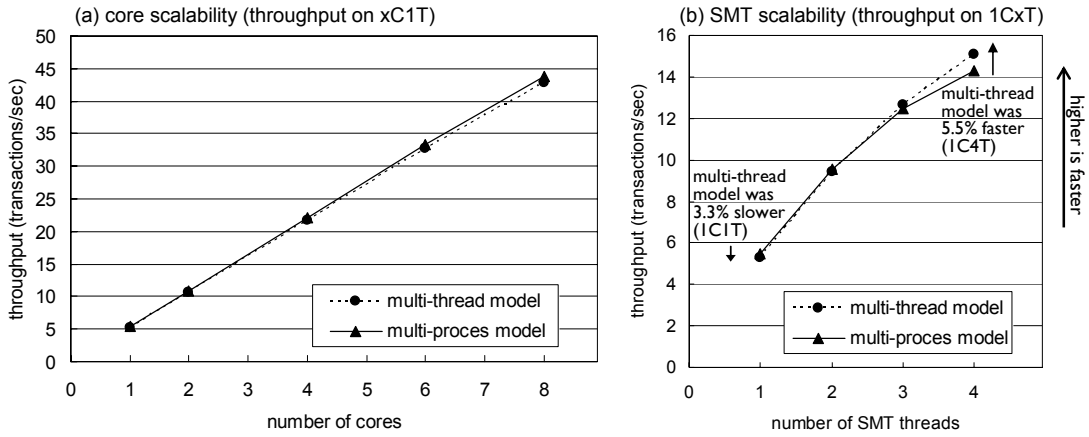


Figure 7. (a) Core scalability using one thread/core and (b) SMT scalability using multiple SMT threads on one core for MediaWiki on Niagara.

0.86% for the multi-thread model with 8-KB pages when using 4threads on 1 core (4C1T).

Though we do not show these results, the multi-process model achieved up to 4.2% better core scalability due to the smaller number of DTLB misses when we used 8-KB pages for the Java heap, while the multi-thread model showed up to 3.4% better core scalability when we used 4-MB pages.

IV. PERFORMANCE RESULTS FOR A PHP WORKLOAD

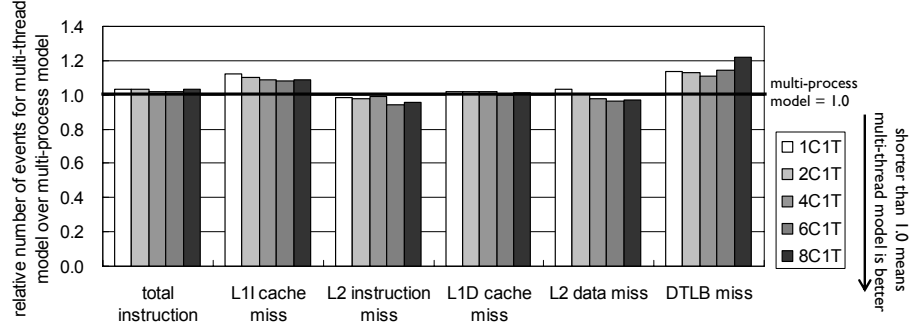
A. System Setup

To confirm that our observations shown in Section III are not specific to Java workloads, we tested a real-world server workload, MediaWiki [15], written in PHP. MediaWiki is a wiki server program developed by the Wikimedia foundation for the Wikipedia online encyclopedia. We imported 1,000 articles from a Wikipedia database dump (<http://download.wikimedia.org/>). We configured MediaWiki to use memcached-1.2.1 running on the same machine to cache the results of the database queries. We measured the throughput while reading randomly selected articles. We used PHP-5.2.1 [16] for the runtime, lighttpd-1.4.13 for the HTTP server, and mysql-4.1.20 for the database server. We also installed the APC-3.0.14 extension

to the PHP runtime, which enhances the performance of PHP applications by caching the compiled intermediate code. The database server and the client emulator ran on separate machines. The number of PHP runtime instances (process or thread) was calculated from the number of hardware threads used for the evaluation by multiplying by 1.5 to hide the latency to access the backend database and fully utilize the processor. Hence we used 48 processes or threads when we used all 32 hardware threads in the processor. The emulated clients paused for three seconds as ‘thinking time’ between each request. The number of emulated clients was set for the highest throughput that allowed the average response time to stay under 2.0 seconds.

Though the default configuration of the original PHP runtime uses the multi-process model, it also supports the multi-thread model by enabling a compile-time option called ZTS. Figure 6 shows the structure of the multi-thread model and the multi-process model for the PHP runtime. Each PHP runtime instance handles independent requests and there is no communications among the instances. The original ZTS implementation used the pthread library for thread-local memory. To improve the performance, we modified the PHP runtime to use the thread-local construct (`__thread`) provided by the Sun C compiler instead of the pthread library, because

(a) using increasing number of cores (up to 8 cores)



(b) using increasing number of SMT threads (up to 4 threads)

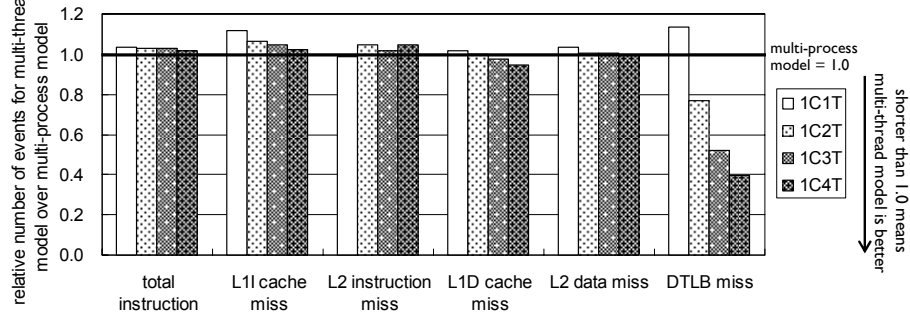


Figure 8. Comparisons of the numbers of instructions and cache misses with multi-thread model against multi-process model for MediaWiki (PHP) on Niagara while (a) increasing the number of cores and (b) increasing the number of SMT threads.

the pthread library has larger overhead for better portability. We also modified the PHP runtime to use multiple UNIX domain sockets to communicate with the HTTP server. We did not need to change the MediaWiki source code to run the program in the multi-thread model. We explicitly specified a memory page size of 4 MB for the heap area by using the MPSS (Multiple Page Size Support) library.

B. Core Scalability and SMT scalability

Figure 7 shows the core scalability and SMT scalability of MediaWiki. For the core scalability, both models achieved almost linear speedup with the increase in the number of cores used, although the multi-process model outperformed the multi-thread model from 2.0% to 3.3% regardless of the number of cores. The PHP runtime was originally optimized for a multi-process model and thus enabling the multi-threading support adds overhead for extra indirections to access global variables. The multi-process model was faster when using only one SMT thread per core due to this additional overhead. For the SMT scalability, however, the multi-thread model showed better scalability with increasing numbers of SMT threads and outperformed the multi-process model when using three or four SMT threads. The multi-thread model was slower by 3.3% when using only one SMT thread (1C1T) due to the thread safety overhead but it became faster than the multi-process model by 5.5% when using four SMT threads (1C4T) because of the benefit of reduced DTLB misses and cache misses in the multi-thread model. This advantage was smaller than the advantage of

9.5% for Java workloads shown in Figure 2. This was partially because the JVM inherently supported multiple threads and thus there was no additional overhead due to the multi-threading support.

Figure 8 shows the relative number of instructions and cache misses per transaction for the multi-thread model over the multi-process model in MediaWiki. In Figure 8(a), the number of instructions per transaction was slightly larger for the multi-thread model than for the multi-process model due to the additional instructions to make the runtime thread-safe. Supporting thread-safety also increased the number of cache misses. By increasing the number of cores, the gap between the multi-thread model and multi-process model in the number of L2 cache misses was slightly reduced due to the smaller memory footprint of the multi-thread model. When using four or more cores, the multi-thread model generated fewer L2 cache misses in spite of the additional indirect accesses for thread safety. In contrast, only the gap in the number of DTLB misses was increased by increasing the number of cores. When multiple software threads running on separate cores shared the same memory page, each thread was able to use only a part of the page, so the number of the required memory pages was increased, causing an increase in the DTLB misses. As a result of the increased DTLB misses and fewer L1 and L2 cache misses with more cores, the multi-thread model achieved scalability almost comparable to the multi-process model. In Figure 8(b), the relative number of DTLB misses was drastically decreased with the use of more SMT threads. Due to this large reduction in DTLB

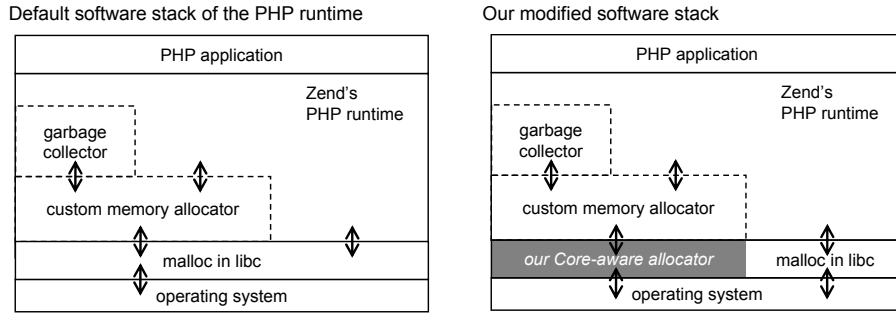


Figure 9. Software stacks of the PHP runtime with and without our core-aware allocator. Only the part shown in gray was modified to use the core-aware allocator.

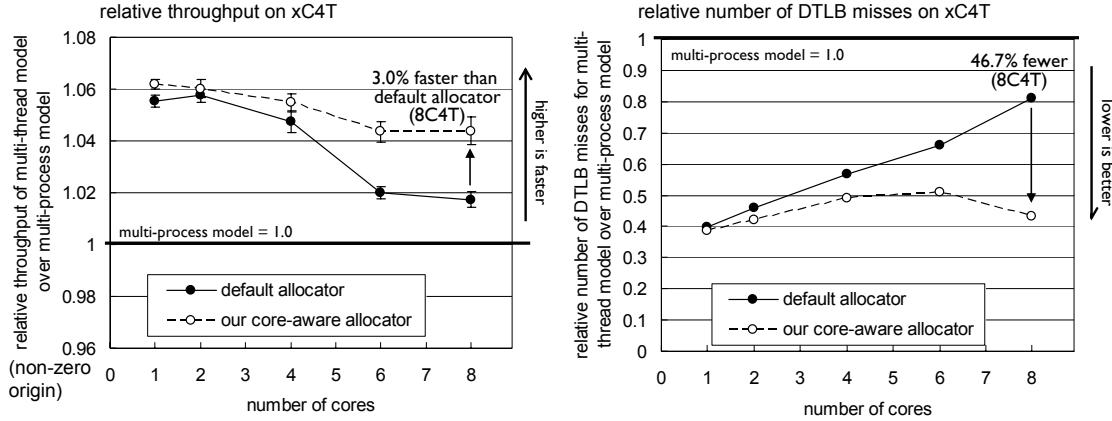


Figure 10. Relative throughput and the number of DTLB misses of the multi-thread model over the multi-process model with and without our core-aware allocator for MediaWiki on Niagara. Error bars show 95% confidence intervals.

misses, the multi-thread model achieved much better SMT scalability than the multi-process model, while its core scalability was roughly equal to the multi-process model.

V. CORE-AWARE MEMORY ALLOCATION

A. Motivation

As described in the previous sections, the multi-thread model achieved higher performance on each core with multiple SMT threads. However we found that the performance advantage of the multi-thread model was reduced as the number of cores increased. For example, the performance advantage of the multi-thread model over the multi-process model for MediaWiki was only 1.7% when using all eight cores, whereas it was 5.5% on one core. The difference in the DTLB miss ratio was also decreased by increasing the number of cores. As shown in Figure 8(a), the multi-thread model tended to increase the number of DTLB misses by using more cores. One naive technique to avoid such increases in DTLB misses is to use multiple processes and bind each process to a specific core. However, this naive technique often suffers from imbalanced load among the cores and increased memory use.

B. Implementation

To maximize the performance of the multi-threaded programs on many SMT processor cores by further reducing

the number of DTLB misses, we implemented a memory allocator in the multi-threaded PHP runtime. We call this allocator the core-aware allocator. The core-aware allocator tracks which core executed each requester (a software thread calling the allocator) and allocates memory blocks from the same memory page for each core. The idea of a memory allocator being aware of the physical location of the requester is not new. For example, most of the memory allocators used in modern UNIX systems support local memory allocation in NUMA systems to avoid costly remote memory accesses. Tikir and Hollingsworth [11] also implemented a NUMA-aware memory allocator in a JVM. With our core-aware allocator, we seek to show that a memory allocator that is aware of the physical location of the requester at a finer granularity can improve the performance of the running programs even on a non-NUMA system. Though our allocator is not specialized for SMT processors, the effect of the DTLB misses is typically larger on processors with many SMT threads and thus our allocator is most suitable for exploiting multi-core SMT processors.

The current PHP runtime uses a custom memory allocator. The custom memory allocator requests fixed-size (256 KB by default) memory chunks to the underlying memory allocator. The custom memory allocator requests a new chunk if a request cannot be handled in the already allocated memory chunks. The underlying memory allocator can use malloc in libc (the default) or an mmap system call in UNIX-based

systems. We implemented our allocator as an underlying memory allocator and the PHP runtime calls it instead of the malloc in libc. We did not modify the custom memory allocator of the PHP runtime itself. Figure 9 illustrates our modifications to the PHP runtime.

Our allocator obtains at startup time a 256-MB memory block for each core using an mmap system call. We explicitly specify 4 MB as the page size. Our allocator treats the 256-MB memory block as a pool of 1024 memory chunks of 256 KB each because each request from the custom allocator in the PHP runtime is always a multiple of 256 KB. When our allocator receives a request, it returns a memory chunk from the pool for the core which is running the requester. To identify the core with the requester, we use the lwpsinfo file for the requester thread in the proc file system. To avoid excess overhead in accessing this file system, we only access the lwpsinfo file once per Web transaction, when the PHP runtime receives a new transaction from the HTTP server. If an operating system frequently moves threads to different cores, we would need to increase the frequency of these location checks.

C. Performance with our core-aware allocator

Figure 10(a) shows how our core-aware allocator implemented in the multi-threaded PHP runtime improved the throughput. We conducted the measurement four times and averaged the results. The reduction in the performance of the multi-thread model was successfully avoided with our core-aware allocator. The multi-thread model with the core-aware allocator outperformed the multi-process model by 4.4% and the multi-thread model with the default allocator by 3.0% when we used all of the hardware threads in Niagara (8C4T). To examine the causes of the differences with and without the core-aware allocator, Figure 10(b) shows the relative number of DTLB misses over the multi-process model with and without the core-aware allocator. The core-aware allocator avoided increase in the number of DTLB misses with increases in the number of cores. The reduction in DTLB misses with the core-aware allocator when using eight cores (8C4T) was 46.7%. These results showed that a memory allocator should track which core is executing a program to maximize the performance of the running programs on multi-core SMT processors.

VI. SUMMARY

In this paper, we studied the interactions between the parallelization model and the processor architecture aspects of a multi-core SMT processor. We evaluated the performance and the detailed architecture-level statistics of programs implemented in the multi-process model and the multi-thread model. Our results showed that both models achieved almost comparable core scalability, whereas the multi-thread model achieved much better SMT scalability and higher performance. We showed that the DTLB misses were the key to the differences between the core scalability

and the SMT scalability.

Although the multi-thread model showed significant advantages on one core with multiple threads, the gain was reduced as the number of cores increased. To maximize the performance on the multi-core SMT processors, we implemented a memory allocator for a multi-threaded PHP runtime to reduce DTLB misses on multi-core SMT processors. Our allocator was aware of the core that is running the requester and allocates memory from the same memory page for each core. This reduced the DTLB misses by 46.7% compared to the multi-threaded PHP runtime without our allocator, and performance was improved by 3.0%. These results showed that a memory allocator should track which core is executing a program to maximize the performance of the running programs on SMT processors.

REFERENCES

- [1] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the Annual International Symposium on Computer Architecture*, pp. 392–403, 1995.
- [2] M. Shah, J. Barren, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, B. Sana, D. Sheahan, L. Spracklen, A. Wynn. UltraSPARC T2: A highly-treaded, power-efficient, SPARC SOC. In *Proceedings of the IEEE Solid-State Circuits Conference*, pp. 22–25, 2007.
- [3] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded SPARC Processor. *IEEE Micro* 25(2), pp. 21–29, March/April 2005.
- [4] R. Kalla. POWER7: IBM's Next Generation POWER Microprocessor. *A Symposium on High Performance Chips* 21, 2009.
- [5] R. Singhal. Inside Intel Core Microarchitecture (Nehalem). *A Symposium on High Performance Chips* 20, 2008.
- [6] R. Iyer, M. Bhat, L. Zhao, R. Illikkal, S. Makineni, M. Jones, K. Shiv, D. Newell. Exploring Small-Scale and Large-Scale CMP Architectures for Commercial Java Servers. In *Proceedings of IEEE International Symposium on Workload Characterization*, pp. 191–200, 2006.
- [7] D. Kaseridis and L. K. John. CMP/CMT Scaling of SPECjbb2005 on UltraSPARC T1. *Workshop on Computer Architecture Evaluation using Commercial Workloads*. 2007.
- [8] J. H. Tseng, H. Yu, S. Nagar, N. Dubey, H. Franke, P. Pattnaik, H. Inoue, and T. Nakatani. Performance studies of commercial workloads on a multi-core system. In *Proceedings of IEEE International Symposium on Workload Characterization*, pp. 57–65, 2007.
- [9] Y. Ueda, H. Komatsu, and T. Nakatani. Scalability Study of a Java Application Server on Two Multi-core Systems. *Workshop on Computer Architecture Evaluation using Commercial Workloads*, 2008.
- [10] M. Ohara, P. Nagpurkar, Y. Ueda, and K. Ishizaki. The Data-centricity of Web 2.0 Workloads and its Impact on Server Performance. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pp. 133–142, 2009.
- [11] K. Ishizaki, S. Daijavad, and T. Nakatani. Analyzing and Improving Performance Scalability of Commercial Server Workloads on a Chip Multiprocessor, In *Proceedings of the IEEE International Symposium on Workload Characterization*, pp. 217–226, 2009.
- [12] M. M. Tikir and J. K. Hollingsworth. NUMA-Aware Java Heaps for Server Applications. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pp. 108, 2005.
- [13] Standard Performance Evaluation Corporation. SPECjbb2005. <http://www.spec.org/jbb2005/>.
- [14] Standard Performance Evaluation Corporation. SPECjvm2008. <http://www.spec.org/jvm2008/>.
- [15] Wikimedia Foundation. MediaWiki. <http://www.mediawiki.org>.
- [16] The PHP Group. PHP: Hypertext Preprocessor. <http://www.php.net/>.