CMP/CMT Scaling of SPECjbb2005 on UltraSPARC T1

Dimitris Kaseridis and Lizy K. John
Department of Electrical and Computer Engineering
The University of Texas at Austin
{kaseridi, ljohn}@ece.utexas.edu

Abstract

The UltraSPARC T1 (Niagara) from Sun Microsystems is a new multi-threaded processor that combines Chip Multiprocessing (CMP) and Simultaneous Multi-threading (SMT) with an efficient instruction pipeline so as to enable Chip Multithreading (CMT). Its design is based on the decision not to focus the performance of single or dual threads, but rather to optimize for multithreaded performance in a commercial server environment that tends to feature workloads with large amounts of thread level parallelism (TLP). This paper presents a study of Niagara using SPEC's latest Java server benchmark (SPECjbb2005) and provides some insight into the impact of the architectural characteristics and chip design decisions. According to our study, we found that adding an additional hardware thread per core can achieve up to 75% of the performance of adding an additional core. Moreover, we show that any additional hardware thread beyond two and up to the limit of four can achieve approximately 45% of the improvements of a single core.

1. Introduction

For the past several years, microarchitecture designers have been using process scaling and aggressive speculation as the means for improving processors performance. It is a common belief in the research community that the performance growth of microarchitectures will slow substantially due to poor wire scaling and diminishing improvements in clock rates as the semiconductor feature size scales down [1]. For this reason, recent trends in microprocessor architectures have diverged from previous well-studied designs, basically driven by the increasing demands for performance along with power efficiency. Such approaches move away from single core architectures and experiment with concepts based on multiple cores and multithreaded designs [2, 3]. Sun's UltraSPARC T1 [4, 5] is such a processor.

A processor like Niagara is designed for favoring overall throughput by exploiting thread level parallelism. Niagara implements a combination of chip multi-processing (CMP) and simultaneous multi-threading (SMT) design to form a chip multi-threaded organization (CMT). Sun's architects selected an eight core CMP with each core being able to handle four hardware threads, which results in an equivalent 32-way machine, virtually presented to the operating system. From this design point, we can analyze its performance and quantify the benefits that additional cores and/or additional hardware threads have on a Java server workload benchmark like SPECjbb2005 [6] which intents to stress processor and cache implementation along with system's scalability and memory hierarchy performance.

2. Description of UltraSPARC T1 Architecture

In November 2005, SUN released its latest microprocessor, the UltraSPARC T1, formerly known as "Niagara" [4, 5]. The distinct design principle of the processor is the decision to optimize Niagara for multithreaded performance. This approach promises increased performance by improving throughput, the total amount of work done across multiple threads of execution. This is especially effective in commercial server applications such as web services and databases that tend to have workloads with large amounts of thread level parallelism (TLP).

The Niagara architecture, shown in Figure 1, consists of eight simple, in-order cores, or individual execution pipelines, with each one of them being able to handle four active context threads that share the same pipeline, the *Sparc pipe*. This configuration allows the processor to support up to 32 hardware threads in parallel and simultaneously execute eight of them per CPU cycle. The striking feature of the processor that allows it to achieve higher levels of throughput is that the hardware can hide the memory and pipeline stalls on a given thread by effectively scheduling the other threads in a pipeline with a zero-cycle switch penalty.

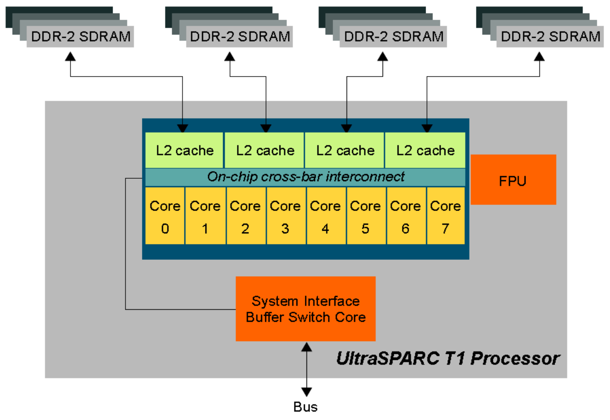

Figure 1. UltraSPARC T1 architecture [5]

In addition to the 32-way multi-threaded architecture, Niagara contains a high-speed, low-latency crossbar that connects and synchronizes the 8 on-chip cores, L2 cache memory banks and the other shared resources of the CPU. Due to this crossbar, the UltraSPARC T1 processor can be considered a Symmetric Multiprocessor system (SMP) implemented on chip. Furthermore, in order to achieve low levels of memory latency and be able to provide the required volume of data to all of the 8 cores operate in parallel, SUN included four dual-data rate 2 (DDR2)

DRAM on-chip channels. This memory configuration achieves a maximum bandwidth in excess of 20GBytes/s, and a capacity of up to 128 Gbytes.

The above characteristics allow the UltraSPARC T1 processor to offer a new thread-rich environment for developers and promise higher level of throughput by hiding memory and pipeline stalls and favoring the execution of multiple threads in parallel. Taking everything into consideration, the processor has the potential to scale well with applications that are throughput oriented, particularly web, transaction processing, and Java technology services. The classes of applications that Niagara is expected to scale up are: multi-threaded applications, multi-process applications, Java technology-based applications and multi-instance applications.

3. Description of Performance counters and tools to extract data

The Niagara processor includes in its design a set of hardware performance counters [5, 7] that allows the counting of a series of processor events per hardware thread context which include cache misses, pipeline stalls and floating-point operations. Statistics of processor events can be collected in hardware with little or no overhead, making these counters a powerful tool for monitoring an application and analyzing its performance. The Niagara and the Solaris 10 operating system provide a set of tools for configuring and accessing these counters. The basic tools that we used were: *cpustat* and *cputrack* [8]. Table 1 contains the microarchitectural events that all of theses tools allow us to monitor along with a small description of each one of them. In addition to these events *mpstat* and *vmstat* [8] tools of Solaris 10 can provide higher level of statistics for individual hardware-threads and virtual memory, respectively.

Table 1. Events of instrumented performance counters

Event Name	Description
Instr_cnt	Number of completed instructions.
SB_full	Number of store buffer full cycles
FP_instr_cnt	Number of completed floating-point instructions
IC_miss	Number of instruction cache (L1) misses
DC_miss	Number of data cache (L1) misses for loads
ITLB_miss	Number of instruction TLB miss trap taken.
DTLB_miss	Number of data TLB miss trap taken (includes real_translation misses).
L2_imiss	Number of secondary cache (L2) misses due to instruction cache requests.
L2_dmiss_ld	Number of secondary cache (L2) misses due to data cache load requests.

In addition to the above tools, we used Solaris' *psrset* and *psradm* [8] administrator tools for changing the configuration of T1 processor. These tools allow the user to create custom sets of hardware context thread slots and specifically bind processes for execution on them. Having this flexibility, we were able to execute processes in an 8-way and 16-way like execution by creating sets of one and two hardware context thread slots for each of the 8 different cores, respectively. By using all of the aforementioned tools, we were able to monitor the operation of the UltraSPARC processor in all of the different phases of the SPECjbb2005 benchmark execution.

4. SPECjbb2005 overview

Enterprise Java workloads constitute an important class of applications that feature the creation of numerous small objects with relatively short life-time. These short-lived objects, which most of the times process various kinds of transactions with databases, create a significant pressure on the garbage collection subsystem of the Java Virtual Machine (JVM). SPECjbb2005 (Java Server Benchmark) [6] is the latest iteration from SPEC that attempts to model a self contained, three-tier system that includes clients, server and database elements. In this paper we used SPECjbb2005 as the benchmark for studying Niagara.

In SPECjbb2005 [6], each client is represented by an individual thread that sequentially executes a set of operations on a database emulated by a collection of Java objects. The database contains approximately 25MB of data. Every instance of a database is considered as an individual warehouse. During the execution of the benchmark the overall number of the individual warehouses is scaled so as to allow more clients to connect simultaneously to the available warehouses and therefore more threads execute in parallel. The performance measured by SPECjbb covers CPUs, caches and the memory hierarchy of the system without generating any I/O disk and network traffic since all of the three tiers of the modeled system are implemented in the same JVM. Furthermore, the reported score is a throughput rate metric that is proportional to the overall number of transactions that the system is able to serve in a specific time interval and therefore reflects the ability of the server to scale to larger number of executed threads.

5. Methodology

The UltraSPARC T1 used for this study was configured as shown in the table below:

Table 2. Experimental Parameters

Parameter	Value
Operating System	SunOS 5.10 Generic_118833-17
CPU Frequency	1000 MHz
Main Memory Size	8 Gbytes DDR2 DRAM
JVM version	Java(TM) 2 build 1.5.0_06-b05
SPECjbb Execution Command	Java -Xmx2560m -Xms2560m -Xmn1536m -Xss128k -XX:+UseParallelOldGC -XX:ParallelGCThreads=15 -XX:+AggressiveOpts -XX:LargePageSizeInBytes=256m -cp jbb.jar:check.jar spec.jbb.JBBmain -propfile SPECjbb.props

In order to quantify the benefits of using additional cores and/or additional threads on a SPECjbb2005 kind workload, we conducted a series of experiments using different system configurations. With the help of Solaris' processor sets (*psrset tool* [8]) we created three different configurations using in any time all of the eight cores and varying only the number of activated hardware threads per core. The three selected configurations were: a) eight cores using one thread per core (equiv. to an 8-way CMP), b) eight cores, using two threads per core (equiv. to a 16-way CMT with each core having a virtual 2-way SMT configuration) and finally c) eight cores using 4 hardware threads per core (equiv. to a 32–way CMT). To be able to analyze the microarchitectural impact of the previous configurations, we used both the microarchitectural performance counters provided by the processor and the information provided by the operating system tools.

6. Results and Analysis

In the first experiment we evaluate the performance benefits of adding additional cores in a CMP configuration for the case of SPECjbb. For doing so, we configured our processor according to the first configuration, which has only one active hardware context thread per core, and therefore models a CMP system. Since SPECjbb gradually increases the overall number of warehouses during a regular run, we were able to analyze the effect of increasing the actual number of cores being used. Figure 2 shows the SPECjbb score for gradually increasing number of warehouses for the first case study.

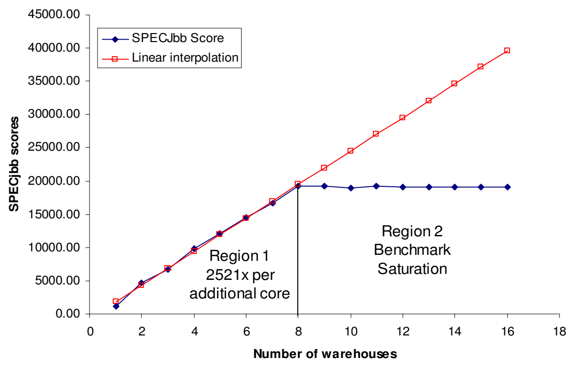

Figure 2. SPECjbb scores for 8 cores x 1 thread per core

As we scale the number of warehouses there is the possibility that the OS could schedule multiple threads per core or even migrates threads during the execution of the benchmark between different available cores. In order to analyze the behavior of the threads throughout the execution of the benchmark, we measured the involuntary context switches that took place in every core. Notice that for the case of Niagara, only one thread is executed in every core per cycle. When a memory or pipeline stall takes place in a core, the execution is switched to another available thread so as to hide the stall penalty and improve the overall throughput. For the case of Niagara, this kind of context switches is called voluntary context switches. In addition to that, the scheduler can force the switch between threads when more than one thread is available for execution and the one executing has exceeded a specific time interval. These switches are named involuntary context switches. When using only one thread per core, according to the first used configuration, an involuntary context switch forces the executing thread out of the core in order to execute another available thread. Therefore, for the cases of one up to eight warehouses in which cases there are at most eight available threads, the number of involuntary switches can show if a thread migrates between the available cores throughout the benchmark execution. This number of involuntary context switches for the cases of one up to eight warehouses was found extremely low. On the other hand, when the number of warehouses goes beyond eight and therefore more than one thread has to be executed on some of the cores, this number of switches increases significantly. This provides confidence that for the cases of one up to eight warehouses, the threads do not migrate between cores, allowing us in this way to study the effectiveness of using additional cores. Moreover, such a behavior shows that all of the cores are equally utilized in the cases of more than eight warehouses as benchmark scales up to more warehouses and therefore more than two threads have to be executed in some of the cores.

From Figure 2 we can clearly see that there are two distinct regions: The first one covers the cases between 1 to 8 warehouses while the second one covers the cases from 9 up to 16 warehouses. In the first region we can see that the addition of an extra workload thread (allocated to a new core since every core uses only one hardware context thread) forms a linear relation with the SPECjbb score. By analyzing the data we found that each additional core adds approximately 2520 additional SPECjbb points to the overall score. To verify this behavior, we run the benchmark only on one core using one thread, which is equivalent to an in-order, single thread processor. Figure 3 shows the result of the execution. From this figure we can see that the score for a small number of warehouses is constant and close to 2500 which is in agreement with the previous regression analysis. Moreover, since the score is a rate metric of completed transactions in a given time interval, the addition of extra threads although decreases the completed transactions per thread, does not have a big effect on the score since the core is almost fully utilized and completes the same overall transactions along all concurrent executed warehouses. This is an indication that the memory hierarchy is capable of keeping the core busy by providing data to all the threads and the small decrease in score, according to our measurements using *mpstat* tool [8], is due to the gradually increasing number of involuntary context switches. We should note that the initial improvement from the first up to the third warehouse is due to the Just-in Time (JIT) compilation of JVM which translates the workloads bytecode into native machine code and therefore accelerates the execution of each Java thread.

The second region of Figure 2 shows a constant behavior when executing more than one thread per core and context switching takes place. This behavior shows that there is very small performance degradation due to the necessary context switches that have to take place in order to execute the additional threads. Table 3 contains the characteristics throughout the execution of benchmark for the first configuration as were captured using the provided performance counters. We should note that throughout the analysis, the IPC calculated as an average IPC across all the cores that is the actual number of instruction executed per core divided by the number of CPU cycles and the number of overall used cores.

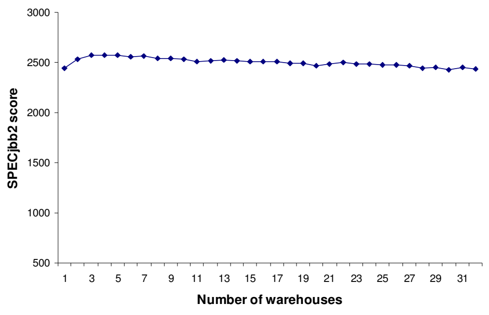

Figure 3. SPECjbb scores for 1 core x 1 thread per core

Table 3. Performance Counters for 8 cores x 1 thread

Characteristic	Average Value
IPC	0.27
IC_misses	1.37 %
DC_misses	3 %
DTLB_misses	0.0001 %
L2_imisses	0.01 %
L2_dmisses_ld	0.66%
SB_full	21%

From Table 3 we can see that all the percentages of the performance counters are very low which shows that the processor is able to feed the eight cores with the required data and can follow the benchmark memory footprint. This behavior was expected since every core handles only one thread and there is no resource sharing per core for fetching the needed data, allowing in that way the executed threads to take advantage of the high memory bandwidth the processor provides. In all of the cases the value of gathered events follows the same pattern as the one shown in Figure 4 for the case of DC misses. Initially we have a high percentage of misses, which are caused by the compulsory misses and the execution of JIT. After that initial area, we could see cases that are different from the average value only when the benchmark finishes a set of warehouses and moves to the next one by loading a new set. This static behavior is another indication that in the case of the second region of Figure 2, the benchmark score saturates because each core runs in full speed and not due to a bottleneck in one of the memory or communication resources.

In order to examine the impact of SMT and the benefits of adding additional hardware context threads, we configured Niagara as a 16-way CMT machine by enabling two hardware context slots per core. Figure 5 shows the measured scores for each individual number of warehouses for the case of having two hardware threads per core. In this case, we can separate the figure in three different regions of interest. The first one, which is actually the same region with the first region of the previous analyzed case, has in this case a benefit of approximately 2540 SPECjbb marks per inserted core in the final reported scores. The second region is of more interest in this experiment since in this region, which covers the cases of 8 to 16 warehouses, by increasing the number of warehouses we gradually increase the number of hardware threads that are used by the benchmark to service the request for the warehouses. Following the same analysis as before we found that every additional hardware thread yields to almost 1960 SPECjbb points, which is lower than the benefit we have for the case of every additional core of region 1. Since in this case each core handles more than one thread the communication infrastructure is shared between the two threads and it is expected to have a small degradation in the benefit of adding an additional hardware thread. Despite this degradation, we can see that every additional hardware thread can achieve 75% of the performance benefit of adding an additional single core. Additional hardware threads achieve this gain with significantly less area and power requirements compared to using additional cores.

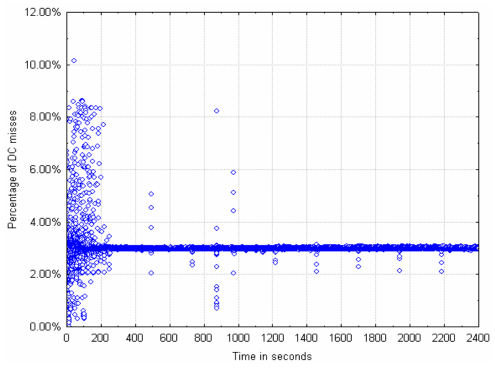

Figure 4. DC misses for the case of 8 cores x 1 thread

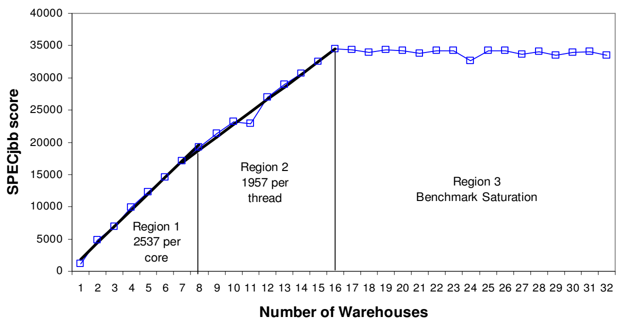

Figure 5. SPECjbb scores for 8 cores x 2 threads per core

In Table 4 we can see the average performance characteristics throughout the execution of the benchmark for the second studied configuration. As for the case of Table 3, the values of the performance counters of Table 4 are relatively low which shows that the characteristics captured by the counters do not significantly affect the performance of the benchmark. Therefore, the only reason for the benchmark saturation in the region 3 of Figure 5 is again the fact that all of the cores are fully utilized. During region 2 we measured an increase in the involuntary context switches per core, which is actually the number of times each simple core that can execute only one hardware thread per cycle freezes the currently executing hardware thread and switches the execution to a new available one. This increased number of context switches was expected since each simple core can now execute two threads and have to switch between them, ending up in this way in a smaller benefit per extra used thread. Moreover the performance data show that adding an extra thread can effectively keep a high throughput for twice the number of warehouses resulting in almost twice the score.

Table 4. Performance Counters for 8 cores x 2 threads case

Characteristic	Average Value
IPC	0.44
IC_misses	1.6 %
DC_misses	3.3 %
DTLB_misses	0.0001 %
L2_imisses	0.01%
L2_dmisses_ld	0.75 %
SB_full	20.2 %

The final case of study shown in Figure 6, concerns the execution of the benchmark using the processor without any restrictions that is using 8 cores with 4 hardware threads activated per core. Following the same analysis we can divide the SPECjbb score curve of figure 6 in 5 distinct regions with the first two cases being the same as the previously analyzed one. The third region in this case shows the SMT benefits of adding more than two hardware threads per core. In this case our analysis showed that each additional thread yields to approximately 1150 SPECjbb score points. That is 45% of the score benefit of region 1 for adding an additional core. Therefore from this figure we can see that by using more than two threads per core we gradually have diminishing returns in terms of SMT efficiency of using more hardware threads. Especially for the case of higher number of warehouses the performance counter analysis along with the profiling of each cores showed that each core is highly utilized and has fewer opportunities to hide latency through zero delay switching between threads ready for execution.

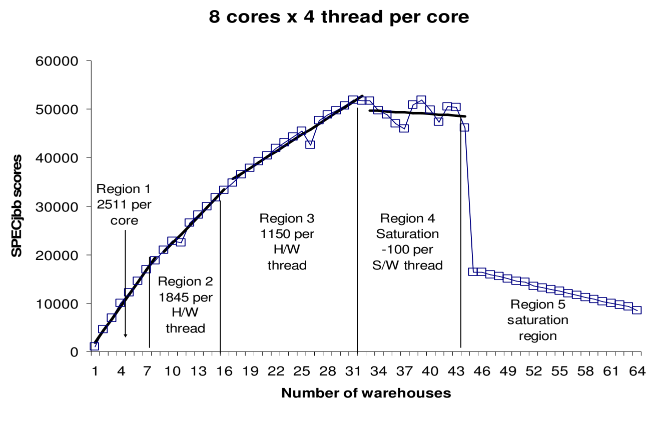

Figure 6. SPECjbb scores for 8 cores x 4 threads per core

Regions 1 up to 3 show the same behavior as in the previously analyzed cases. The reason that performance counter statistics are slightly increased for these regions is due to the philosophy of the execution of the benchmark. SPECjbb2005 follows a ramp up procedure during which the initial set of warehouses are executed for only 30 seconds [6]. On the other hand the statistic for the previous cases concern the overall execution that includes the execution in the cases after each peak performance where the benchmark execution saturates and each set of warehouses is executed for 4 minutes [6].

Table 5. Performance Counters for 8 cores x 4 threads case

Characteristic	Region 1	Region 2	Region 3	Region 4	Region 5
IPC	0.21	0.54	0.55	0.67	0.43
IC_misses	3.9 %	1.9 %	1.9 %	2.1 %	2.2 %
DC_misses	3.4 %	3.4 %	3.4 %	3.5 %	2.7 %
DTLB_misses	0.0001%	0.001%	0.013	0.01 %	0.06 %
L2_imisses	0.048 %	0.01%	0.04 %	0.02 %	0.2 %
L2_dmisses_ld	0.8%	0.78 %	0.8 %	0.79 %	0.55 %

Region 4 represents the area were the benchmark is saturated. In this region every additional software thread causes degradation in the overall score due to the fact that the processor has to execute more threads than the available hardware context threads can handle simultaneously. In this case, by using mpstat, we saw that more threads cause the invocation of more context switches between the threads already assigned and the one waiting to be scheduled. As in the previous cases, in this region the utilatization of the cores is kept high since there are many available for execution threads, which is reflected by the increased average number of IPC in this region.

Region 5 shows a big degradation in the overall score, which is mainly caused by the high number of threads available for execution. In addition to that, in region 4 we can observe a couple of dips in the score. After analyzing the performance counters along with the available system-level information provided by OS and JVM, we concluded that this big gap in score along with the dips of region 4 is due to the invocation of the garbage collector (GC) of JVM. We profiled the operation of GC and saw that after the point of 44 warehouses the GC invokes a lot of times. Every time GC invokes, JVM freezes the execution of the Java thread and therefore the actual number and duration of GC invocation in seconds is inversely proportional to the overall performance of execution of threads. Each transaction in SPECjbb creates short-lived objects that after the completion of their operation persist in heap memory until the GC takes action and frees the occupied memory space. The more warehouses, the more short-lived data created in the time interval of the execution of a set of warehouses. In the case of 38 and 41 warehouses the number of threads is high enough to create a large number of short-lived data, pushing in that way the usage of heap memory close to 100%. In order to handle this situation, JVM calls GC several times so as to free memory space. The point of 44 simultaneous warehouse hits the limit that the temporal data reaches the maximum limit of heap many times through the execution of the warehouses and therefore the GC is called multiple time. After that point GC calls are significantly increased and take more time to complete. To be more specific, our measurements showed that beyond the point of 44 warehouses the execution of GC consumes almost one third of the four minutes execution time interval of SPECjbb [6]. Moreover, after this specific point, every execution of GC last almost 6 seconds when in the case of the previous regions GC duration is limited to 0.2 seconds. Such a radical change in memory mapping is shown by the increased value of DTLB_misses in region 5 with L2_dmisses being on average unchanged. This significant consumption of time in GC in order to keep the used heap memory in its limits is reflected by the big degradation in SPECjbb score. Notice that the limits of regions 4 and 5 depend on the JVM heap size and the available main memory of the system under study. Therefore, the previous limits mainly concern the specific system that we used for our study and its configuration and cannot summarize the scaling limits of the UltraSPARC T1 in general.

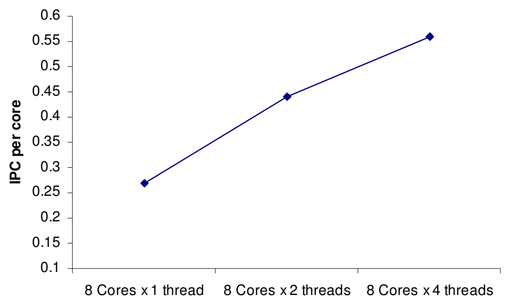

Figure 7. IPC of three configurations

Figure 7 shows the average IPC per core, measured for all of the three different configurations used during the previous analysis when executing SPECjbb. As we can see from the graph, the simplest SMP configuration of every core using two threads gives on average 1.8x speedup over the CMP configuration and when every core is configured as a SMT with four hardware threads we can see an average 1.27x and 2.3x speedup over the 2-way SMT per core and the single-threaded CMP, respectively.

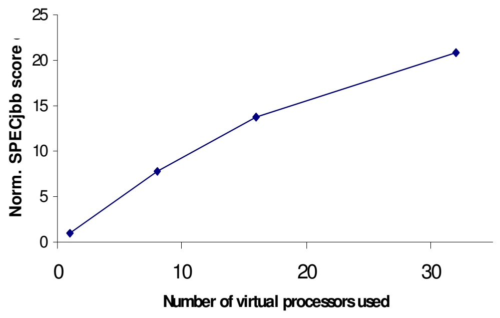

Figure 8. Best case SPECjbb score speedup

Finally, Figure 8 shows the SPECjbb score speedup over the number of virtual processors used. The first point represents the case of using one core and only one thread per core that is a single-threaded, in-order processor and the next three points represent the three different configurations we analyzed throughout the previous analysis section. This speedup is an indication of the top performance we can gain from Niagara and its scalability when switching between the different configurations. According to this figure, using all of the hardware context threads of all the cores can achieve a speedup of almost 20x the performance of the single thread.

6. Conclusions

In this paper we analyzed the impact of the architectural characteristics and chip design decisions of the Niagara CMT processor, which is designed for favoring overall throughput by exploiting thread level parallelism in combination with multiple cores. Through our experiments we quantify the benefits of using additional cores and/or threads on a well-known Java server workload. Our experiments showed that Niagara can achieve 75% of the performance improvement of adding an additional simple core by simple adding only an additional hardware thread per core. Moreover, we showed that adding more than two hardware threads per core we can achieve a 45% of the improvement for adding a single threaded core.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable comments. This research was supported in part by NSF grant 0429806. We would like to thank Sun Microsystems for providing us an UltraSPARC T1 (Niagara) machine to conduct this study.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, D. Burger, “Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures”, Annual International Symposium on Computer Architecture, 2000, VOL 27, pages 248-259.

[2] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, “The case for a single-chip multiprocessor”, In Proceedings of the Seventh international Conference on Architectural Support For Programming Languages and Operating Systems, October, 1996. ASPLOS-VII. pp 2-11

[3] L. Spracklen , S. G. Abraham, “Chip Multithreading: Opportunities and Challenges”, Proceedings of the HPCA ’05, pp. 248-252

[4] P. Kongetira, K. Aingaran and K. Olukotun, “NIAGARA: A 32-way Multithreaded SPARC processor”, IEEE Micro, Volume 25, Issue 2, March-April 2005 Page(s):21 - 29 2005

[5] UltraSPARC T1 Supplement to the UltraSPARC Architecture2005 http://opensparc.sunsource.net/specs/UST1UASuppl-current-draft-HP-EXT.pdf

[6] SPECjbb2005, http://www.spec.org/jbb2005/

[7] A. Singhal, A. J. Goldberg, “Architectural Support for Performance Tuning: A Case Study on the SPARCcenter 2000”, Proceedings of the 21st Annual International Symposium on Computer Architecture, Chicago, IL, April, 1994, pp. 48-59.

[8] Solaris 10 Reference Manual, http://docs.sun.com/app/docs/prod/solaris.10