

Measuring Operating System Overhead on CMT Processors

Petar Radojković¹, Vladimir Cakarevic¹, Javier Verdú², Alejandro Pajuelo², Roberto Gioiosa¹, Francisco J. Cazorla¹, Mario Nemirovsky^{1,3}, Mateo Valero^{1,2}

¹ Barcelona Supercomputing Center (BSC)

² Universitat Politècnica de Catalunya (UPC)

³ICREA Research Professor

{petar.radojkovic, vladimir.cakarevic, roberto.gioiosa, francisco.cazorla, mario.nemirovsky}@bsc.es
{jverdu, mpajuelo, mateo}@ac.upc.edu

Abstract

Numerous studies have shown that Operating System (OS) noise is one of the reasons for significant performance degradation in clustered architectures. Although many studies examine the OS noise for High Performance Computing (HPC), especially in multi-processor/core systems, most of them focus on 2- or 4-core systems.

In this paper, we analyze the major sources of OS noise on a massive multithreading processor, the Sun UltraSPARC T1, running Linux and Solaris. Since a real system is too complex to analyze, we compare those results with a low-overhead runtime environment: the Netra Data Plane Software Suite (Netra DPS).

Our results show that the overhead introduced by the OS timer interrupt in Linux and Solaris depends on the particular core and hardware context in which the application is running. This overhead is up to 30% when the application is executed on the same hardware context of the timer interrupt handler and up to 10% when the application and the timer interrupt handler run on different contexts but on the same core. We detect no overhead when the benchmark and the timer interrupt handler run on different cores of the processor.

1 Introduction

Modern operating systems (OSs) provide features to improve the user experience and hardware utilization. To do this, the OS abstracts real hardware, building a virtual environment, known as a virtual machine, in which the processes execute. This virtual machine makes the user's application believe it is using the whole hardware in isolation when, in fact, this hardware is shared among all processes being executed in the machine. Therefore, the OS is able to offer, through the virtual machine abstraction, features such as multitasking. However, these capabilities come at the cost of overhead in the application execution time.

In this paper, we evaluate the overhead of two wide-

ly used operating systems running on a UltraSPARC T1 processor [1][3], Solaris (version 10) [12] and Linux (Ubuntu 7.10, kernel version 2.6.22-14) [8]. We confirm the results of other studies [10][13], which focused on 2- or 4-core systems, for a CMT processor with a large number of hardware strands. In fact, the OS noise may be negligible on a single machine with few cores/threads, but may become important for parallel applications that have to be synchronized running on a large number of cores, which is the case of HPC applications. For example, assume that a Single Program Multiple Data (SPMD) parallel application is running on a large cluster with thousands of cores. Also, in this example, assume that the application is perfectly balanced and that each process in the parallel application computes for either t_{sec} and then waits for the other processes to complete their iteration before starting a new one. In this scenario, if one of the processes in the application experiences some OS noise its iteration will require more than t_{sec} and since the other processes cannot proceed until the last task reaches the synchronization point, all of the application is slowed down. Moreover, as the number of cores increases, the probability that at least one process in the parallel application experiences the maximum noise during each iteration approaches 1.

Figure 1 reports an experiment from [10] for $t=1ms$ and $t=100\mu s$. The Figure shows how the execution time of a parallel application increases because of the OS noise when the number of cores goes from 2 to 1024. The X-axis presents the number of the cores in the cluster and the Y-axis shows the average latency, i.e., the time the first process that reach the synchronization point suffers because of the OS noise. Figure 1 shows that increasing the number of cores to 1024 affects slowdown of 200% in case when computation phase lasts $100\mu s$ and 20% for $1ms$ computation phase.

Linux and Solaris are both full OSs with many concurrent services and since we run our experiments on a real machine it is not easy to obtain a reference case to compare our results.

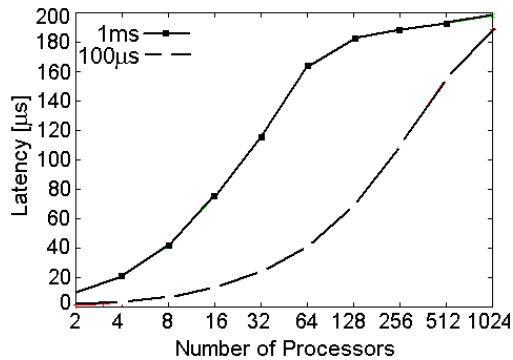


Figure 1. Latency as the function of the number of processors

With our methodology, we obtain the reference case running the experiments on Netra DPS, a light-weight run-time environment [6][7]. A fundamental problem when determining the overhead of the OS is that the OS noise cannot be completely removed from the system when the experiments are performed. Netra DPS is a low-overhead environment that provides less functionalities than Linux and Solaris but introduces almost no overhead. This capability makes Netra DPS ideal for our analysis.

For the evaluation process, we use a set of created synthetic benchmarks that execute only one type of instruction and have only one phase during their execution. This allows us to determine which type of programs are likely to suffer higher overhead from OS activities.

There are two major contributions in this paper. First, we set up a framework based on synthetic benchmarks and the light-weight runtime environment Netra DPS to obtain a baseline execution of those benchmarks without OS noise.

Second, we confirm some of the well-known sources of OS noise for a CMT processor with 32 hardware strands. We show that the process scheduler behavior in Linux and Solaris is significantly different. While in Linux the overhead is homogeneous in all hardware contexts¹, in Solaris the overhead depends on the particular core/strand in which the application runs. Therefore, when we execute our benchmarks in Linux, we detect periodic overhead peaks with a frequency of 250Hz and durations of 15µs to 30µs, which corresponds to timer interrupt handler. We re-execute the benchmarks in different strands of the processor, obtaining the same behavior. This is due to the fact that in Linux the process scheduler executes on every strand of the processor.

There are different performance overheads due to clock tick interrupt when benchmarks run on different strands in Solaris. The reason for this is that Solaris binds the timer interrupt handler to the strand 0 of the logical domain, so no clock interrupt occurs in any strand different from strand 0.

¹In this paper we refer to hardware context also as hardware strand or simply strand.

- When an application runs in strand 0 we observe the highest overhead, regardless of the type of instructions the application executes.
- When the application runs in an other strand in the same core where the timer interrupt handler runs, we also observe some smaller overhead the intensity of which depends on the application's CPI (Cycles Per Instruction): the higher the CPI, the higher the overhead experimented by the application.
- We detect no timer interrupt overhead when applications execute on a core different than the one on which the timer interrupt handler runs.

Hence, high demanding application, sensitive to the overhead introduced by the timer interrupt, running in Solaris, should not run on the first core, definitely not in the first strand. However, in the current version of Solaris, the scheduler does not take this into account when assigning a CPU to a process. Moreover, the scheduler may dynamically change the strand assigned to the application so it is up to users to explicitly bind their applications to specific strands. In our experiments, when an application is not explicitly bound to any strand, Solaris schedules it on the first strand for most of the execution, which leads to performance degradation.

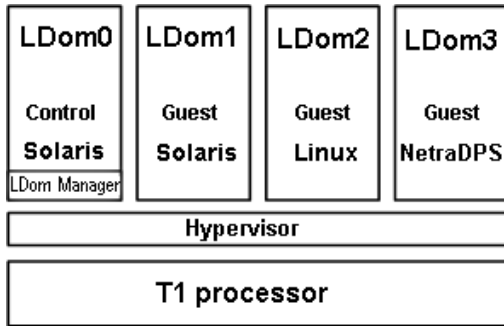
The remaining structure of this paper is as follows: Section 2 describes our experimental setup, Section 3 presents our analysis about the influence of OS ticks to application performance, Section 4 presents related work, and Section 5 presents our conclusions.

2 Experimental Environment

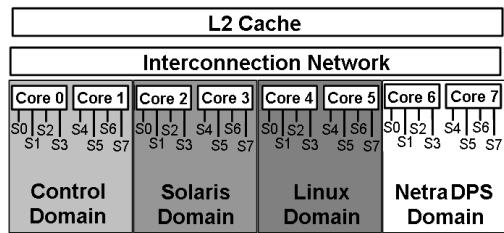
This section describes the hardware environment, the benchmarks, and the software tools that we use in this paper.

2.1 Hardware Environment

In order to run our experiments, we use a Sun UltraSPARC T1 processor running at a frequency of 1GHz, with 16GBytes of DDR-II SDRAM. The UltraSPARC T1 processor is a multithreaded, multicore CPU with eight cores, each of them capable of handling four hardware threads concurrently. Each core is a fine grained multithreading processor, meaning that it can switch between the available threads every cycle. Even if the OS perceives the hardware contexts inside the core as individual logical processors, at a micro architectural level they share the pipeline, the instruction and data L1 caches, and many other hardware resources, such as the integer or the front end floating point unit. Sharing the resources may cause slower per-thread execution time but could increase the overall throughput. Besides the intra-core resources, each hardware context also shares the inter-core resources, i.e., those resources shared between the cores like the L2 cache or the main floating point unit.



(a) Logical view



(b) Mapping of the logical domains onto the cores

Figure 2. LDom setup we use in our experiments

2.2 Logical Domains

The Logical Domains (LDoms) technology allows us to allocate the system’s resources, such as memory, CPUs, and devices, to logical groups and to create multiple, discrete systems, each of which with its own operating system, virtual hardware resources, and identity within a single computer system. In order to achieve this functionality, we use the Sun Logical Domains software [4][5]. LDoms uses the hypervisor firmware layer of Sun CMT platforms to provide stable and low overhead virtualization. Each logical domain is allowed to observe and interact only with those machine resources that are made available to it by the hypervisor.

For our experimentation we create four logical domains (see Figure 2): one control domain (required for handling the other virtual domains) and three guest domains running Solaris, Linux, and Netra DPS, respectively. We allocate the same amount of resources to all guest domains: two cores (8 strands) and 4 GBytes of SDRAM. For each logical domain, strand 0 (s0) is the first context of the first core, strand 1 (s1) is the second context of the first core, strand 4 (s4) is the first context of the second core, and so on.

- **Control domain:** This logical domain manages the resources given to the other domains. On this domain we install Solaris 10 (8/07).
- **Solaris domain:** This domain runs Solaris 10 (8/07).
- **Linux domain:** We run Linux Ubuntu Gutsy Gibon 7.10 (kernel version 2.6.22-14) on **Linux domain**.

	Line	Source code
	001	.inline intdiv_il, 0
	002	.label1:
B	003	sdivx %o0, %o1, %o3
O
D	514	sdivx %o0, %o1, %o3
Y	515	subcc %o2, 1, %o2
	516	bnz .label1
	517	sdivx %o0, %o1, %o3

Figure 3. Main structure of the benchmarks. The example shows the INTDIV benchmark.

- **Netra DPS domain:** Netra DPS allows writing applications in high level language (ANSI C). The peculiarity of this framework is that the scheduling of tasks on the hardware strands is done statically, at compile time. Moreover, only one task can run on each strand, which eliminates the need of a runtime process scheduler.

2.3 Benchmarks

Real, multi-phase, multi-threaded applications are too complex to be used as the first set of experiments because the performance of an application running on a multi-thread/core processor depends on the other processes the application is co-scheduled with. Collecting the OS noise experienced by these applications would be difficult on a real machine running a full-fledged OS. In order to measure the overhead introduced by the OS with our methodology, we need applications that have a uniform behavior so that their performance does not vary when the other applications in the same core change their phase. In order to put a constant pressure to a given processor resource we use very simple benchmarks that execute a loop whose body only contains one type of instruction. By using these benchmarks we can capture overhead due to influence of other processes running in the system, simply by measuring the benchmark’s execution time.

We create a large set of benchmarks, but we present only three of them which we think are representative: integer addition (INTADD), integer multiplication (INTMUL) and integer division (INTDIV), all of them written in assembly for SPARC architectures. All three benchmarks are designed using the same principle (see Figure 3). The assembly code is a sequence of 512 instructions of the targeted type (lines from 3 to 514) ended with the decrement of an integer register (line 515) and a non-zero branch to the beginning of the loop (line 516). After the loop branch (line 516) we add another instruction of the targeted type (line 517) because in the UltraSPARC T1 processor the instruction after the *bnz* instruction is always executed. The assembly functions are inlined inside a C program that defines the number of *iterations* for the assembly loop. The overhead of the loop and the calling code is less than 1% (more than 99% of time processor executes only the desired instruction).

The benchmarks are compiled in the Control domain using the Sun C compiler (Sun C version 5.9), and the same executables are run in the Solaris guest domain. In order to run them in the Linux domain, the object file obtained by the compilation in the Control domain is linked with gcc (version 4.1.3) in the Linux domain.

We compile Netra DPS images in the Control domain with the same Sun C compiler. To ensure the equal application behavior in the Solaris, Linux and Netra DPS domains, we use the same optimization flags in all compilations.

2.4 Methodology

We run each benchmark in isolation, without any other user applications running on the processor. In this way we ensure that there is no influence by any other user process and, therefore, all the overhead we detect is due to the OS activities and the activities due to maintenance of the logical domains environment that we created. To obtain reliable measurements of OS overhead, we use the FAME (FAirly MEasuring Multithreaded Architectures) methodology [17][18]. In [17][18], the authors state that the average accumulated IPC (Instructions Per Cycle) of a program is representative if it is similar to the IPC of that program when the workload reaches a steady state. The problem is that, as shown in [17][18], the workload has to run for a long time to reach this steady state. FAME determines how many times each benchmark in a multi-threaded workload has to be executed so that the difference between the obtained average IPC and the steady state IPC is below a particular threshold. This threshold is called MAIV (Maximum Allowable IPC Variation). The execution of the entire workload stops when all benchmarks have executed as many times as needed to accomplish a given MAIV value. For the experimental setup and benchmarks used in this paper, in order to accomplish a MAIV of 1%, each benchmark must be repeated at least 5 times.

2.5 Tools

In order to measure the execution time of our applications, we read the *tick* register of the SUN UltraSPARC T1 processor. Reading this register returns a 63-bit value that counts strand clock cycles [2].

The Solaris, Linux OSs, and Netra DPS environments provide user support for binding applications to the specific hardware context (virtual processors). In Solaris, to bind process to a virtual processor we use the *processor_bind()* system call invoked in the benchmarks that we execute. The *processor_bind()* function binds a process or a set of processes defined by their *id* to a virtual processor. To bind process to a virtual processor in Linux we use the *sched_setaffinity()* function. The *sched_setaffinity()* function sets the CPU affinity mask of the process denoted by *pid*. The CPU affinity mask, in turn, defines on which of the available processors the process can be executed. In Netra DPS, binding a function to

a virtual processor (strand) is done in a mapping file before compiling the application.

3 Overhead Due to the Job Scheduler

To provide multitasking, the OS introduces the process scheduler. This scheduler is responsible for selecting which process, from those ready to execute, is going to use the CPU next. To perform this selection, the process scheduler implements several scheduling policies. One of them is based on assigning a slice of CPU time, called quantum, to every process to delimit the period in which this process is going to be executed without interruption.

Quantum-based policies rely on the underlying hardware implementation. The hardware has to provide a way to periodically execute the process scheduler to check if the quantum of the running process has expired. To accomplish this, the current processors incorporate an internal clock that raises a hardware interrupt and allows the CPU go into kernel mode. If the quantum of the running process has expired, the process scheduler is invoked to select another task to run. In this section we will show how this hardware interrupt and the process scheduler affect the execution time of processes in Linux, Solaris, and Netra DPS.

Netra DPS applications are bound to strands at compile time and cannot migrate to other strands at run time. For this reason, Netra DPS does not provide a run time scheduler. In order to provide a fair comparison between Linux, Solaris, and Netra DPS, we decided to study the situation in which only one task is ready to execute. In this case, every time the scheduler executes, it just checks that there is no other task ready to execute in that strand. Therefore, the overheads we report here concerning the job scheduler are the lowest that can be observed. Moreover, having more than one application running at the same time will make the study more complex to analyze, as the overhead of the OS on one application could overlap with the activities of the other running applications.

3.1 OS Scheduler Peak Overhead

In order to measure the influence of the process scheduler, we consecutively execute 1000 *repetitions* of every benchmark, where each repetition lasts approximately 100 μ s. The results obtained are the following:

3.1.1 Linux

Figure 4 shows the execution time per repetition of the INTADD benchmark in Linux when it is bound to strand 0. In Figure 4, the X-axis shows the time at which each repetition starts and the Y-axis shows the execution time of the repetition. We observed that the average execution time of repetition is 100 μ s. The important point in this figure is the presence of periodic noise. This noise occurs every 4 milliseconds (250Hz) and corresponds to the interrupt handler associated

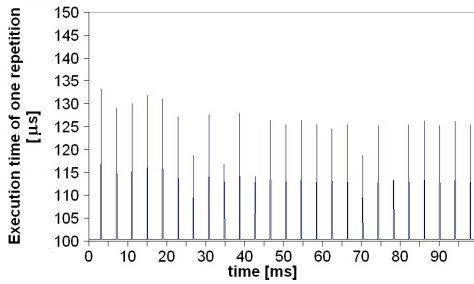


Figure 4. Execution time of the INTADD benchmark when run on Strand 0 in Linux

to the clock tick. Since Linux implements a quantum-based scheduling policy (quantum scheduler with priority), the process scheduler has to be executed periodically to check if the quantum of the process currently being executed is expired or not, or if a higher priority process has waken up. Hence, even if INTADD is executed alone in the machine, its execution is disturbed by that interrupt handler. This makes some repetitions of the benchmark running longer ($123\mu s$), which represents a slowdown of 23%. INTMUL and INTDIV also show those peaks in their execution with the same frequency.

We re-execute the INTADD benchmark in other strands of the processor and obtain the same behavior. In fact, those peaks appear regardless of the strand in which we run the benchmark. This is due to the fact that, in Linux, in order to provide scalability in multithreaded, multicore architectures, the process scheduler is executed in every strand of the processor.

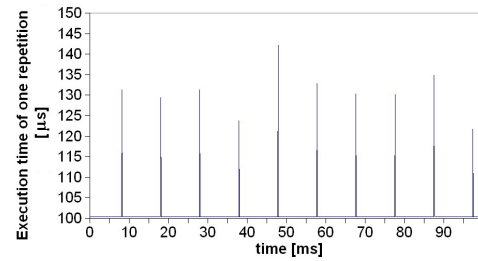
3.1.2 Solaris

Solaris behaves different than Linux. Figure 5 shows the execution time of the INTADD benchmark when it is executed in Solaris. In this case, INTADD is statically bound to strand 0 (Figure 5(a)), strand 1 (Figure 5(b)) and strand 4 (Figure 5(c)).

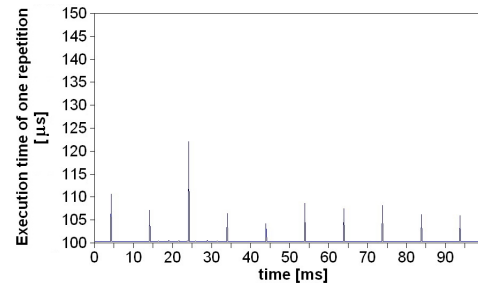
Figure 5(a) shows that, when the benchmark runs in strand 0, the behavior is similar as in Linux. The reason is the same. Since Solaris provides a quantum-based selection policy, the clock interrupt is raised periodically. But, in this case, the frequency of the clock interrupt is 100Hz.

Figure 5(b) shows execution time of INTADD benchmark when it is bound to a strand on the same core where the timer interrupt handler runs. In this case, the peaks are smaller since they are the consequence of sharing hardware resources between two processes running on the same core and not due to the fact that the benchmark is stopped because execution of the interrupt handler and the job scheduler, as it is in the case in strand 0. In Linux we do not detect similar behavior, because the impact is hidden by execution of timer interrupt routine on each strand.

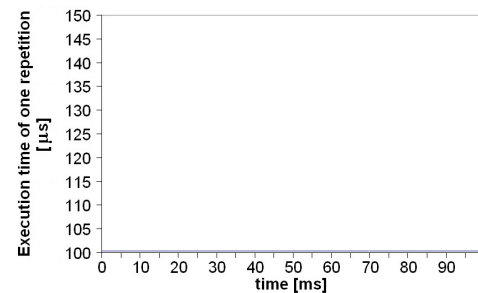
In the UltraSPARC T1 processor, all strands in one core share the resources of that core. One of those shared resources is the fetch bandwidth. Whenever two threads are ready to fetch an instructions, the core distributes the available fetch



(a) Strand 0 (Core 0)



(b) Strand 1 (Core 0)



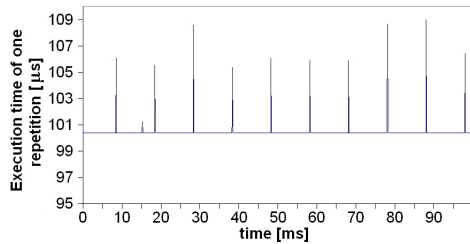
(c) Strand 4 (First strand on Core 1)

Figure 5. Execution time of INTADD in different strands under Solaris

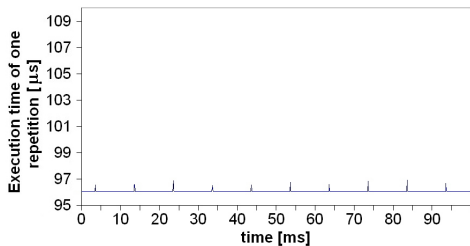
bandwidth among them. In the case of the UltraSPARC T1 processor, the fetch policy is Least Recently Fetched thread. As a consequence, when INTADD runs in strand 1, and no other thread is executed in any other strand in the core, it uses all the available fetch bandwidth. But, when the clock interrupt is raised, and the interrupt handler is executed, the core has to distribute the fetch bandwidth between both processes. This makes INTADD suffer some performance degradation.

When INTADD executes in Strand 4 we do not detect any peaks, see Figure 5(c). Since Solaris binds the timer interrupt handler to strand 0, no clock interrupt is raised in any strand different from strand 0. For this reason, strand 4 (nor any other strand on the same core) does not receive any clock interrupt, which makes the behavior of INTADD stable.

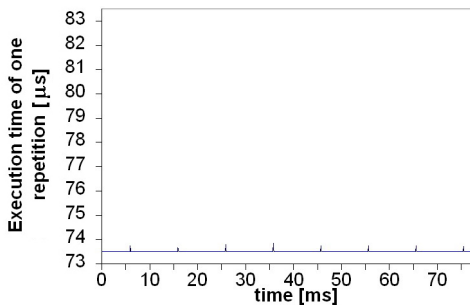
We repeat the experiment for the INTMUL and INTDIV benchmarks. When the tests are performed on strand 0, we detect the same overhead in execution time ($15\mu s$ to $45\mu s$ over the overall behavior) with the same tick frequency (100Hz).



(a) INTADD executed in Strand 1



(b) INTMUL executed in Strand 1



(c) INTDIV executed in Strand 1

Figure 6. Execution time of all benchmarks running on Strand 1 under Solaris

In addition, when the benchmarks are executed in strand 4, the peaks also disappear as it happens to INTADD. But, when the experiments are executed on strand 1, as shown in Figure 6, we notice some differences with respect to the execution time of INTADD (Figure 6(a)), INTMUL (Figure 6(b)) and INTDIV (Figure 6(c)). Note that the scale of Figure 5(c) is different.

In order to clarify this point, we run in Solaris 50,000 repetitions of every benchmark (INTADD, INTMUL, INTDIV) on strand 0 and 1. The results are summarized in Table 1.

We observe that the average overhead is almost the same for all three benchmarks (with a small, 2.2% difference in the worst case) when we run them in strand 0. In this case the overhead is introduced because the benchmark is stopped and the interrupt handler and the OS job scheduler run in strand 0. The overhead is different when we execute threads in strand 1. In this case, the overhead is due to the fact that the benchmark running on strand 1 shares the fetch bandwidth with the timer interrupt handler and job scheduler when they run on strand

Table 1. Average time overhead due clock tick interrupt

Benchmark	CPI	Avg. overhead [μ s]	
		Solaris - strand 0	Solaris - strand 1
INTADD	1	26.415	6.528
INTMUL	11	26.657	1.195
INTDIV	72	26.218	0.823

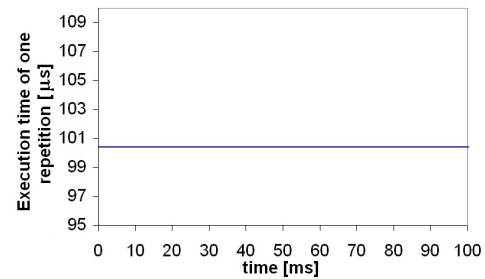


Figure 7. Execution of several INTADD repetitions with Netra DPS in strand 0

0. Given that the use of the fetch bandwidth of a benchmark depends on its CPI, the overhead is different for each benchmark. In fact, the lower the CPI of a benchmark, the more fetch bandwidth it needs, and the higher the effect it suffers when an additional process runs in strand 0.

3.1.3 Netra DPS

Finally, when the INTADD benchmark is executed with Netra DPS, as shown in Figure 7, the peaks do not appear. This is due to the fact that Netra DPS does not provide a runtime scheduler. Threads executed in this environment are statically assigned to hardware strands during compilation. At runtime, threads are always bound to the same strand, so no context switch occurs. In Netra DPS, ticks are not needed for process scheduling which removes the overhead from the benchmark execution. This behavior is present in every strand assigned to Netra DPS.

3.2 OS Scheduler Cumulative Overhead

From the previous section it may seem that the overhead of the OS on the average is small since it only affects few repetitions of the benchmark execution. In fact, process scheduler overhead can only be detected when measurements are taken at a very fine grain, as in the previous examples. But it is important to notice that, when moving to a larger scale, even if no overhead coming from the scheduler can be detected, this overhead accumulates in the overall execution time of the benchmarks. To show this effect, we repeat the experiments but extending the total execution time of each repetition of the benchmarks to 1 second. We make this experiment in Netra DPS and Solaris, running benchmarks on both strand 0 and 4.

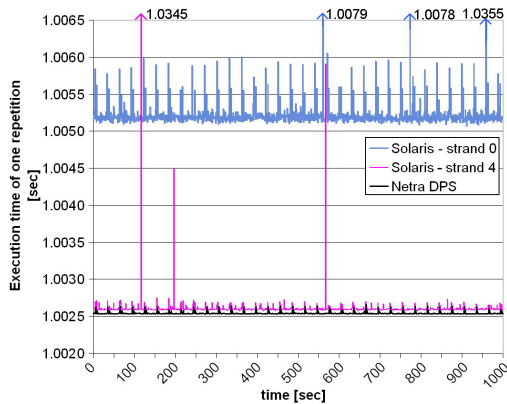


Figure 8. Timer interrupt cumulative overhead in Solaris OS

Figure 8 presents the behavior of the INTADD benchmark. In this Figure, the bottom line corresponds to Netra DPS, whereas the middle and the topmost lines correspond to the benchmark when it is executed with Solaris in strand 4 and strand 0, respectively. The X-axis shows the time at which each repetition starts and the Y-axis describes execution time per repetition.

As shown in Figure 8, Netra DPS, for the reasons explained in the previous section, is the environment that presents the best execution time for the benchmark even when measurements are taken in coarse grain. Small peaks appearing in the execution of INTADD under Netra DPS come from some machine maintenance activities due to the Logical Domain manager. Unfortunately, this overhead noise cannot be evicted when other logical domains (Control domain, Linux, and Solaris domains) are present on the machine. Our experiments reveal that maintenance of logical domains causes a global overhead noise in all strands of the processor similar to those shown in Figure 8 for Netra DPS. In order to confirm that those peaks are neither due to execution application in Netra DPS nor the hypervisor activities, we re-execute benchmarks in Netra DPS without LDoms and we detect no peaks in execution time.

The second best time in Figure 8 relates to the execution of the benchmark with Solaris in strand 4. Notice that the overhead peaks (the smallest ones) caused by the LDom management layer are also present.

Finally, the benchmark presents its worst execution time when it is executed with Solaris in strand 0 (topmost line in Figure 8). This overhead comes from the cumulation of the clock interrupt overheads.

Figure 9 draws the distribution of the samples (for Netra DPS, Solaris-strand 4 and Solaris-strand 0) shown previously in Figure 8. For Figure 9, the X-axis describes execution time, whereas the Y-axis shows the number of samples (repetitions) that have a given execution time. In this Figure, samples make three groups from right to left. The first group, ranging from 1.005×10^9 to 1.006×10^9 cycles, covers the samples of the

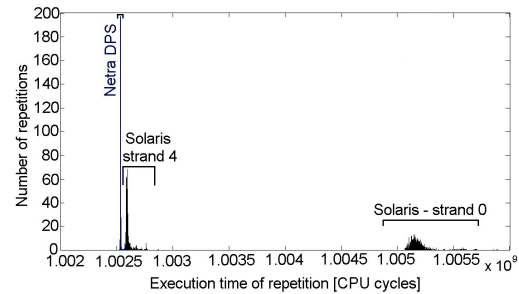


Figure 9. Sample distribution in Netra DPS and Solaris

execution of the INTADD benchmark with Solaris in strand 0. The second group, from 1.0026×10^9 to 1.0028×10^9 , is related to the execution with Solaris in strand 4. And, finally, the third group corresponding to Netra DPS is centered in the execution time point of 1.0025×10^9 cycles.

Two major conclusions can be drawn from Figure 9. First, as previously seen in Figure 8, Netra DPS is the configuration that presents the smallest variance in the execution of all repetitions. All repetitions last for 1.0025×10^9 cycles. Second, Solaris in both strand 0 and strand 4 presents higher variance. The range of variation is on average 0.0001×10^9 and 0.003×10^9 cycles when a benchmark runs on strand 4 and strand 0, respectively.

Figure 8 and Figure 9 lead us to the conclusion that Netra DPS is a very good candidate to be taken as a baseline for measuring the overhead of operating systems since it is the environment that clearly exhibits the best and most stable benchmark execution time.

Stable execution time makes Netra DPS an ideal environment for parallel applications running on large number of cores, as it is in the case of HPC applications.

4 Related Work

The OS system noise as a cause of application performance degradation has been very well explored in the literature. Many studies tried to quantify, characterize and reduce effects of system noise in application execution. Gioiosa et al. [10] and Nataraj et al. [13] point out that timer interrupts and local timer interrupts are the reason of most frequent noise events causing over 50% of the overall system noise in Linux. Even if system noise affects one-compute node applications insignificantly as it is presented in [10][11][14][15], it can cause significant performance degradation for fine-grained applications. [11][14][15][16] show that low intensity, but frequent system noise due to OS clock interrupts is the reason for significant performance degradation of fine-grained applications being the great obstacle to their scalability. Our contribution in the field is exploring behavior of OS services on multicore, multithreaded, processor (UltraSPARC T1) presenting ways to decrease and even completely avoid overhead due to clock tick interrupt in Solaris.

In [9], authors study how applications executed at the same time on an UltraSPARC T1 processor affect each other and use this information to make optimal co-schedules. Instead, in this paper we focus on measuring OS overhead. In order to do so, we run only one application at the same time. Having more than one application running at the same time will make the study more complex, if not impossible, as the overhead of the OS on one application can be confused with the impact of the other running applications.

5 Conclusions

The sources of performance overhead in the current OS have been deeply analyzed in the literature, with a strong focus on multichip computers. However, to our knowledge, this is the first work studying system overhead on a CMT chip. In particular, we compare execution time of several benchmarks on an UltraSPARC T1 processor running Linux and Solaris OSs, and the Netra DPS environment. In the study presented in this paper we use Netra DPS as an environment in which application scheduling is done at compile time. This property makes Netra DPS a good environment to minimize overhead due to mentioned system noise sources, making it a good environment for highly parallel systems in which this service is not required.

In order to study OS overhead, we bind benchmarks on different strands in every studied OS. In Linux, we detect homogeneous overhead due to the clock interrupt handler and the job scheduler in all strands. This is due to the fact that in Linux the process scheduler executes in every strand of the processor. In Solaris we observe different performance overhead due to clock tick interrupt depending on the strand a benchmark runs. The reason behind this is that Solaris executes clock tick interrupt handler on strand 0 of its logical domain. Thus, the highest overhead is introduced when the application runs on strand 0, regardless of the application properties. Less overhead is shown when the application is executed on another strand in the same core. The impact depends on the CPI of the application: the lower the CPI is, the higher the overhead. Finally, we do not detect any overhead when the application and clock tick interrupt handler run on different cores.

The conclusions we obtain from this paper come from single-threaded applications. Even so, they may be applied in scheduling of parallel applications running on a large number of processors where the slowdown suffered by any thread, for example due to a wrong scheduling decision, will likely affect the execution time of the entire application.

Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain under contracts TIN- 2004-07739-C02-01, TIN-2007-60625, the HiPEAC European Network of

Excellence and a Collaboration Agreement between Sun Microsystems and BSC. The authors wish to thank the reviewers for their comments, Jochen Behrens, Gunawan Ali-Santosa and Ariel Hendel from Sun for their technical support, and Bob Guarascio, also from Sun, for editing support.

References

- [1] *OpenSPARCTM T1 Microarchitecture Specification*, 2006.
- [2] *UltraSPARC Architecture 2005*, 2006.
- [3] *UltraSPARC T1TM Supplement to the UltraSPARC Architecture 2005*, 2006.
- [4] *Beginners Guide to LDoms: Understanding and Deploying Logical Domains*, 2007.
- [5] *Logical Domains (LDoms) 1.0 Administration Guide*, 2007.
- [6] *Netra Data Plane Software Suite 1.1 Reference Manual*, 2007.
- [7] *Netra Data Plane Software Suite 1.1 User's Guide*, 2007.
- [8] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Inc., 2006.
- [9] D. Doucette and A. Fedorova. Base vectors: A potential technique for microarchitectural classification of applications. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, in conjunction with ISCA-34, 2007.
- [10] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare. Analysis of system overhead on parallel computers. In *Proceedings of the Fourth IEEE International Symposium on Signal Processing and Information Technology*, 2004.
- [11] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, and M. Roberts. Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003.
- [12] R. McDougall and J. Mauro. *SolarisTM Internals*. Sun Microsystems Press, 2007.
- [13] A. Nataraj, A. Morris, A. D. Malony, M. Sottile, and P. Beckman. The ghost in the machine: Observing the effects of kernel operation on parallel application performance. In *International Conference for High Performance Computing, Networking, Storage and Analysis SC07*, November 2007.
- [14] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asc q. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003.
- [15] D. Tsafirir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *Experimental computer science on Experimental computer science*, pages 171–182, 2007.
- [16] D. Tsafirir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick. System noise, os clock ticks, and fine-grained parallel applications. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 303–312, 2005.
- [17] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. Analysis of system overhead on parallel computers. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 305–316, 2007.
- [18] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. Measuring the Performance of Multithreaded Processors. In *SPEC Benchmark Workshop*, 2007.