Measuring Operating System Overhead on Sun UltraSPARC T1 Processor

Petar Radojković*, Vladimir Cakarevic*, Javier Verdú†, Alejandro Pajuelo†, Roberto Gioiosa*, Francisco J. Cazorla*, Mario Nemirovsky*‡, Mateo Valero*†

*Barcelona Supercomputing Center (BSC)
†Universitat Politècnica de Catalunya (UPC)
‡ICREA Research Professor

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, especially in multi-processor/core systems, most of them focus on 2- or 4-core systems.

In this study, we analyze sources of OS noise on a massive multithreading processor, the Sun UltraSPARC T1. We compare results, measured in Linux and Solaris, with the results provided by a low-overhead runtime environment that introduces almost no overhead in applications' execution time. 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 as 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.

KEYWORDS: Operating Systems; Overhead; Multicore multithreaded processors;

1 Introduction

Modern operating systems (OSs) provide features to improve the user experience and the hardware utilization. To do this, the OS abstracts the real hardware to processes and makes the user's application believe it is using the whole hardware in isolation when, in fact, this hardware is shared among all processes running in the machine. Hence, the OS is able to offer features like multitasking or virtual extensions of the available physical memory.

However, these capabilities come at the cost of overhead in the application execution time and should be taken into account especially when running real-time applications, since in some cases that overhead is significant. For example, the OS job scheduler is also a process that competes with other user process for the available hardware resources. So, the overhead introduced by the process scheduler should be taken into account when deriving conclusions about application performance.

In this study, we evaluate the overhead of the process scheduler by comparing two widely-used operating systems, Solaris and Linux, and the Netra Data Plane Software (Netra DPS) environment [net07]. We run all experiments on a real multicore multithreaded processor, Sun UltraSPARC T1.

2 Environment

In order to run our experiments, we use a Sun UltraSPARC T1 [t106] 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.

In order to run different operating systems on a single UltraSPARC T1 processor, we use *Logical Domains*, low-overhead virtual machine in Hypervisor layer. Logical Domain technology allows the allocation of system's resources (memory, CPUs, and devices) into logical groups and the creation of multiple discrete systems, each with its own operating system, resources and identity within a single computer system. For our experiments we create four logical domains: one control domain (running Solaris 10 [MM07]) and three guest domains also running Solaris (Solaris 10), Linux (Ubuntu Gutsy Gibon 7.10, kernel version 2.6.22-14) [BC06], and Netra DPS (version 1.1). We allocate the same amount of resources (two cores and 4GBytes of memory) to all guest domains in order to set up the environment in such a way that provides the maximum fairness in comparison of OSs running on the domains.

In order to measure the overhead of the OS, we use applications that present an uniform behavior. So, all execution time peaks that could appear in their execution come directly from the OS. To put a constant pressure to a given processor resource we use very simple benchmarks, written in assembly for SPARC architectures, that execute a loop whose body contains only one type of instruction. We create large set of benchmarks, but due to space limit, we present only one of them (comprised of integer add instructions) which we think is representative. Using this benchmark, we capture the overhead due to the influence of other processes running in the system, simply by measuring the benchmark execution time.

To measure the overhead due to the OS (management activities), we run each benchmark in isolation, without any other user application running on the physical processor. Hence, we can ensure that there is no influence by any other user process.

3 Results

To provide multitasking, the OS provides a process scheduler. This scheduler is responsible of 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, delimiting the period in which this process is going to be executed without interruption.

Quantum-based policies rely on the underlying hardware. The hardware has to provide a mechanism to periodically execute the process scheduler to check if the quantum of the process being executed has expired. To accomplish this, current processors incorporate internal clocks that raise hardware interrupts every constant periods of time. The process scheduler handles these interrupts to perform the process selection and thus multitasking.

In this section, we show how this hardware interrupt and the process scheduler affect the execution time of processes in Linux, Solaris, and Netra DPS. We design the benchmark to have constant execution time (100museconds) when it runs in isolation (no other processes

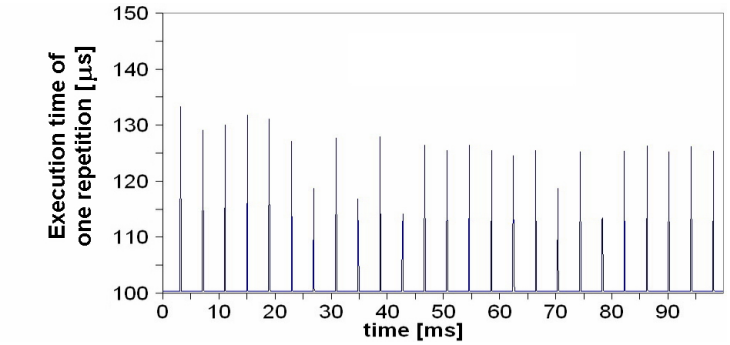


Figure 1: Execution time of the benchmark when run on Strand 0 in Linux

running on the processor). In order to measure the influence of the process scheduler, we consecutively execute 1000 repetitions of the benchmark and measure the execution time of each repetition.

Figure 1 shows the execution time per repetition of the benchmark in Linux when it is bound to strand 0. In Figure 1, the X-axis shows the time at which each repetition starts and the Y-axis shows the execution time of the repetition. When benchmarks run in the **Linux domain**, we notice the presence of periodic peaks in execution time. These peaks occur every 4 milliseconds (250Hz) and correspond to the interrupt handler associated 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 active process is expired or not, or if a higher priority process has waken up. Hence, even if the benchmark is executed alone in the machine, its execution is disturbed by that interrupt handler. This makes that some repetitions of the benchmark have a slowdown of over 20%.

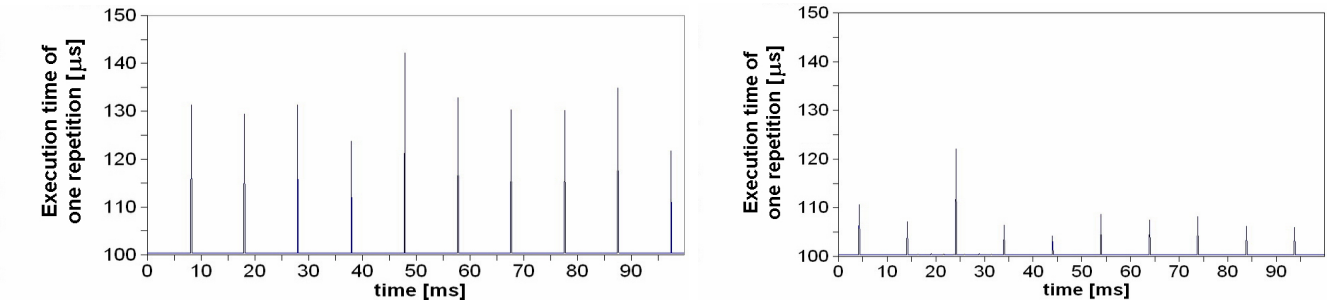
We re-execute benchmarks in other strands of the processor and obtain the same behavior. In fact, those peaks appear regardless of the strand in which we execute the benchmark. The reason for this behavior is that in Linux, to provide scalability in multithreaded and/or multicore architectures, the process scheduler is executed in every context of the processor.

Solaris behaves different then Linux. The results for benchmark when it executes in Solaris, are presented on Figure 2. Figure 2(a) shows that, when the benchmark executes in strand 0, the behavior is similar as in Linux. The reason is the same. Since Solaris provides a quantum-base selection policy, the clock interrupt is raised periodically. But, in this case, the frequency of the clock interrupt is 100Hz.

Figure 2(b) shows execution time of 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 (e.g. Instruction Fetch Unit) between two processes running on the same core and not due the fact that the benchmark is stopped because execution of the interrupt handler and the job scheduler, as when the benchmark runs on strand 0. In Linux we do not detect similar behavior, because the impact is hidden by execution of timer interrupt routine on each strand.

When benchmark executes in strand 4, we do not detect any peaks. Since Solaris binds the process scheduler to the strand 0, no clock interrupt is raised in any strand different from strand 0. For this reason, strand 4 does not receive any clock interrupt, which makes the behavior of benchmark stable. Furthermore, since the benchmark running on strand 4 does shares only global processor resources (such as L2 cache, IO, or Floating Point Unit) with task running on strand 0 (as they are in different cores), it is not affected by the resource sharing with the interrupt handler and the job scheduler, when they are executed.

Finally, when the benchmark is executed on the **Netra DPS domain** 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. Since



(a) Strand 0 (Core 0) (b) Strand 1 (Core 0)

Figure 2: Execution time of benchmark in different strands in Solaris

there are no thread migration from one strand to another and threads are neither stalled nor resumed, no runtime scheduler is needed. 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.

4 Conclusions

The sources of performance overhead in the current OSs 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 Chip MultiThreading (CMT) processor. In particular, we compare execution time of several benchmarks on an UltraSPARC T1 processor running Linux and Solaris OSs, and the Netra DPS environment. As the baseline, we use Netra DPS, 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 because, in Linux, the process scheduler executes in every strand of the processor. In Solaris, we observe different performance overhead because of clock tick interrupt depending on the strand a benchmark runs. This is because 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. Less overhead is shown when the application is executed on another strand in the same core. Finally, we do not detect any overhead when the application and clock tick interrupt handler run on different cores.

Acknowledgements

This work has been supported by the Ministry of Science and Technology of Spain under contracts TIN-2007-60625, the HiPEAC European Network of Excellence and a Collaboration Agreement between Sun Microsystems and BSC. The authors wish to thank Jochen Behrens and Aron Silverton from Sun Microsystems for their technical support.

References

[BC06] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Inc., 2006.

[MM07] R. McDougall and J. Mauro. *Solaris^{TM} Internals*. Sun Microsystems Press, 2007.

[net07] *Netra Data Plane Software Suite 1.1 Reference Manual*, 2007.

[t106] *OpenSPARC^{TM} T1 Microarchitecture Specification*, 2006.