Journal of Advances in Science and Technology

Vol. IV, Issue No. VIII, February-2013, ISSN 2230-9659

AN EVALUATION UPON IMPACT OF OPERATING SYSTEM EXECUTION ON SIMULTANEOUS MULTITHREADED PROCESSOR

AN INTERNATIONALLY INDEXED PEER REVIEWED & REFEREED JOURNAL

www.ignited.in

Journal of Advances in Science and Technology
Vol. IV, Issue No. VIII, February-2013, ISSN 2230-9659

An Evaluation upon Impact of Operating System Execution on Simultaneous Multithreaded Processor

Dr. Shailendra Singh Sikarwar[1] Mahesh Bansal[2]

[1]Assistant Professor, P. G. V. College, Gwalior

[2]Assistant Professor, P. G. V. College, Gwalior

Abstract – This paper presents the first analysis of operating system execution on a simultaneous multithreaded (SMT) processor. While SMT has been studied extensively over the past 6 years, previous research has focused entirely on user-mode execution. However, many of the applications most amenable to multithreading technologies spend a significant fraction of their time in kernel code. A full understanding of the behavior of such workloads therefore requires execution and measurement of the operating system, as well as the application itself.

To carry out this study, we (1) modified the Digital Unix 4.0d operating system to run on an SMT CPU, and (2) integrated our SMT Alpha instruction set simulator into the SimOS simulator to provide an execution environment. For an OS-intensive workload, we ran the multithreaded Apache Web server on an 8-context SMT. We compared Apache's user- and kernel-mode behavior to a standard multiprogrammed SPECInt workload, and compared the SMT processor to an out-of-order superscalar running both workloads. Overall, our results demonstrate the micro architectural impact of an OS-intensive workload on an SMT processor and provide insight into the OS demands of the Apache Web server. The synergy between the SMT processor and Web and OS software produced a greater throughput gain over superscalar execution than seen on any previously examined workloads, including commercial databases and explicitly parallel programs.

----------♦----------

INTRODUCTION

Simultaneous multithreading (SMT) is a latency-tolerant CPU architecture that executes multiple instructions from multiple threads each cycle. SMT works by converting thread-level parallelism into instruction-level parallelism, effectively feeding instructions from different threads into the functional units of a wide-issue, out-of-order superscalar processor. Over the last six years, SMT has been broadly studied and Compaq has recently announced that the Alpha 21464 will include SMT. As a general-purpose throughput enhancing mechanism, simultaneous multithreading is especially well suited to applications that are inherently multithreaded, such as database and Web servers, as well as multiprogrammed and parallel scientific workloads.

This paper provides the first examination of (1) operating system behavior on an SMT architecture, and (2) a Web server SMT application. For server-based environments, the operating system is a crucial component of the workload. Previous research suggests that database systems spend 30 to 40 percent of their execution time in the kernel, and our measurements show that the Apache Web server spends over 75% of its time in the kernel. Therefore any analysis of their behavior should include operating systems activity.

Operating systems are known to be more demanding on the processor than typical user code for several reasons. First, operating systems are huge programs that can overwhelm the cache and TLB due to code and data size. Second, operating systems may impact branch prediction performance, because of frequent branches and infrequent loops. Third, OS execution is often brief and intermittent, invoked by interrupts, exceptions, or system calls, and can cause the replacement of useful cache, TLB and branch prediction state for little or no benefit. Fourth, the OS may perform spin-waiting, explicit cache/TLB invalidation, and other operations not common in user-mode code. For these reasons, ignoring the operating system (as is typically done in architectural simulations) may result in a misleading characterization of system-level performance. Even for applications that are not OS-intensive, the

performance impact of the OS may be disproportionately large compared to the number of instructions the OS executes.

For SMT, a functional processor and operating system do not yet exist. In lieu of these, we extended the SimOS-Alpha infrastructure, adding an Alpha-based SMT core as the instruction execution engine. SimOS is a simulator detailed enough to boot and execute a complete operating system; in the case of the Compaq Alpha, SimOS executes PAL code as well. We also modified the Digital Unix 4.0d operating system to support SMT. This modification is quite straightforward, because Digital Unix is intended to run on conventional shared-memory multiprocessors and is therefore already synchronized for multithreaded operation.

As the first study of OS behavior in an SMT environment, our goal is to answer several basic questions. First, how would previously reported results change, if at all, when the operating system is added to the workload? In particular, we wish to verify the IPC results of previous studies to see whether they were overly optimistic by excluding the OS. For these studies, we used a multiprogrammed workload consisting of multiple SPECInt benchmarks. Second, and more important, what are the key behavioral differences at the architectural level between an operating-system-intensive workload and a traditional (low-OS) workload, both executing on SMT? For example, how does the operating system change resource utilization at the microarchitecture level, and what special problems does it cause, if any, for a processor with fine-grained resource sharing like SMT? For this question, we studied one OS-intensive application, the widely used Apache web server, driven by the SPEC Web benchmark. We compared the Apache workload and the SPECInt workload to study the differences in high-OS and low-OS usage. Third, how does a Web server like Apache benefit from SMT, and where does it spend its time from a software point of view? This analysis is interesting in its own right, because of the increasing importance of Web servers and similar applications. We therefore present results for Apache on an out-of-order superscalar as well as SMT. Overall; our results characterize both the architectural behavior of an OS-intensive workload and the software behavior (within the OS) of a key application, the Apache Web server.

BACKGROUND OF SMT

The history of SMT research can be broken into two distinct areas: architectures based on dynamic superscalars and those based on other architectures such as VLIW. The earliest work was in the non-superscalar field; the .MARS-M. system and the .Matsushita Media Research Laboratory processor. are examples. The move towards superscalar based designs started with the .Multistreamed superscalar processor.

The model of SMT widely used in recent years [Tullsen98] is based on work carried out at the University of Washington. The Washington design was originally based on a static superscalar architecture [Tullsen95]. The design was evaluated against a single-threaded dynamic superscalar with the same issue width (number of instructions that can be executed per cycle) and found to outperform the dynamic superscalar (which itself outperformed an IMT-style design).

The Washington work moved towards a dynamic, out-of-order, superscalar design when it was found that such an architecture could be made to be multithreaded with only a small cost. The design started with a high-performance out-of-order superscalar design similar in spirit to the MIPS R10000. To support multiple threads multiple program counters were added; these were fetched from on an interleaved basis. Structures that needed to be per-thread, such as retirement and the trap mechanism, were duplicated. Thread tagging support was added to shared data structures, such as the branch target buffer, where the ownership of an entry was not made clear by the renaming of registers. The design supported eight threads which placed a large demand on physical registers therefore they opted to increase the size of the register file and pipeline its access.

OPERATING SYSTEM SUPPORT

Abstracting threads in an SMT processor as logical processors is a convenient way to provide instant backwards compatibility; however, it introduces complications into the otherwise simple action of counting processors.

Many commercial operating systems are licensed for a particular number of processors. An interesting problem with building logical processors on top of physical processors is which level of the hierarchy should be counted for the license. If it is decided to count logical processors then a further complication is caused by the ability to disable threads. Should the licensing count be based on the number of threads enabled, or the total number possible? An argument for the latter case is that a check could be carried out at boot time and the threads re-enabled later. Hyper-Threading is now a standard feature on all new high-end Pentium 4 processors so if a system without SMT is required then the second logical processor on each package must be disabled. A license using the logical processor count is likely to be unfair in such a situation.

The two obvious choices for numbering the logical processors are:

1. Number first by package and then by logical processor within each package,

2. Number through the first logical processor in each package then through the second and so on.

The second method is useful in situations where the operating system is licensed for a particular number of processors and runs on the lowest numbers processors, ignoring the remainder. In the first numbering system this would cause entire physical packages to be ignored while logical processors compete on the active packages. The enumeration of processors can be performed by the OS or BIOS. In the latter case it is important that the OS knows how the BIOS performed the enumeration.

Knowledge of the difference between logical and physical processors is useful for load-balancing and scheduling. In a scenario where a system has two physical packages each of two logical processors, and has two runnable processes, the scheduler has to decide which processors to use and which to leave idle. A scheduler unaware of the processor hierarchy may assign the tasks to the lowest numbered processors; using the Linux enumeration method these would be the two logical processors of the first package. The entire second package would be idle which would not give the highest system throughput. Although Linux 2.4.17 was .Hyper-Threaded aware. it exhibited this problem; later versions were able to support the logical/physical processor distinction.

A PROCESS SCHEDULER FOR SMT PROCESSORS

There are two ways in which an SMT processor could be allocated by a scheduler:

- The processor could be treated as a set of individual, independent logical processors. This requires that the hardware threads are heavyweight, i.e. processes. This interface is provided by Intel's Hyper-Threaded processors.

- An SMT processor could be used to run true multithreaded workloads. In this scenario the physical processor is allocated as a single resource rather than separating out the logical processors. A multithreaded workload written or compiled for this particular SMT processor would be able to exploit its characteristics.

A scheduler that is completely unaware of the hierarchical nature of SMT processors using the logical processor abstraction will not know that resources are shared between .sibling. processors. As described in this paper this can lead to runnable tasks being scheduled on two logical processors of one package while another package remains idle. In addition, such a scheduler misses out on some scheduling flexibility: since logical processors on the same package share much state, particularly caches, a process can be migrated between logical processors with little loss compared to migrating between physical processors. This flexibility is useful when balancing load across processors.

The term *processor affinity* is used to describe a given process having a preference to be scheduled to execute on one or more specified processors. Processor affinity can be influenced by factors such as soft-state built up on one processor, the suitability of a processor for the task or the available resources in a heterogeneous system. Most multiprocessor schedulers use processor affinity in some form.

Cache affinity is a form of processor affinity where the process has an affinity for the soft-state it has built up in the processor caches. Whilst this may seem to be the same as basic processor affinity it is actually a dynamic scheme. If a given process it preempted by a second process that causes data belonging to the first process to be evicted from the cache then the first process' affinity for that cache (and therefore processor) reduces. The extreme is that all data belonging to the first process is evicted. In this case the process has no affinity for any cache/processor so the scheduler can assign it to any processor. Measuring the cache affinity of a process accurately would need potentially costly hardware support but an approximation can be made by counting cache line evictions to estimate process cache footprints.

The concept of cache affinity is relevant to SMT architectures because the caches are shared by the threads. The scheduler need not worry which logical processor within a given physical package it assigns a process to, the view of the cache will be the same.

The SMT specific knowledge useful to scheduling will be of the interactions of processes with certain characteristics. The scheduler must therefore know about the characteristics of currently running, or candidate processes. Such knowledge could be acquired by static inspection of program text segments (the executable code) or dynamic measurement of the running processes.

Static analysis of the programs is beneficial in its cost (a onetime activity) but only provides a limited amount of information; effects such as mis-speculation and cache hit rates are important. These effects could only be obtained off-line through simulation/emulation; it would be just as well to run the code and measure it. Dynamic measurement of the running processes provides more information, not only on the process itself but on how it is interacting with the processes on the other logical processor(s).

Dynamic measurement will incur an overhead for both sampling and evaluating the sampled data.

OPERATING SYSTEM EXECUTION

OS simulation environment - At one level the OS is simply a large program; however, it is unique in having access to low-level hardware resources (e.g., I/O device registers and internal CPU registers) and responding to lowlevel hardware events (e.g., exceptions and interrupts). To simulate the OS thus requires simulating those resources and events. In this work, we built upon the SimOS-Alpha hardware simulation framework, integrating our SMT CPU simulator into SimOS.

This allows us to boot and run the operating system on the simulator and include in our simulation every instruction, privileged or non-privileged, that would be executed on a real CPU. The SimOS environment also executes Alpha PAL code – a layer of software that exists below the operating system itself. PAL code is used, for example, to respond to TLB misses and to handle synchronization within the OS (SETIPL). We also model almost all OS/hardware interactions that affect the memory hierarchy, such as DMA operations and cache flush commands. The one exception is DMA operations from the network interface; although including network-related DMA would double the number of memory bus transactions for the Apache workload (the SPECInt workload doesn't use the network), the average memory bus delay would remain insignificant, since it is currently only 0.25 cycles per bus transaction.

Our studies focus on CPU and memory performance bottlenecks. In the interest of simulation time, we simulate a zero-latency disk, modeling a machine with a large, fast disk array subsystem. However, all OS code to manipulate the disk is executed, including the disk driver and DMA operations. Modeling a disk-bound machine could alter system behavior, particularly in the cache hierarchy.

OS modifications - We execute the Compaq/Digital Unix 4.0d operating system, a (shared-memory) multiprocessor-aware OS. By allowing SMT to appear to the OS as a shared-memory multiprocessor (SMP), the only required changes to the OS occur where the SMT and SMP architectures differ. In the case of the Alpha, these differences are SMT's shared TLB and L1 caches, versus the per-processor TLB and L1 caches of an Alpha SMP. Of these two differences, only the TLB-related OS code required modification.

The OS we execute contains the set of minimal changes required to run Digital Unix on an SMT, but does not explore the numerous opportunities for optimizations. For example, OS constructs such as the idle loop and spin locking are unnecessary and can waste resources on an SMT. (However, in the experiments presented in this paper, idle cycles constituted no more than 0.7% of steadystate CPU cycles, and spin locking accounted for less than 1.2% of the cycles in the SPECInt workload and less than 4.5% of cycles in the Apache workload.) Another possible optimization would be to replace the MP OS process scheduler with an SMT-optimized scheduler. We plan to investigate OS optimizations as future work, but it is encouraging that an SMP-aware OS can be modified in a straight-forward fashion to work on an SMT processor.

CONCLUSION

In this paper, we reported the first measurements of an operating system executing on a simultaneous multithreaded processor. For these measurements, we modified the Compaq/DEC Unix 4.0d OS to execute on an SMT CPU, and executed the operating system and its applications by integrating an SMT instruction-level simulator into the Alpha SimOS environment. Our results showed that:

1. For the SPECInt95 workload, simulating the operating system does not affect overall performance significantly for SMT, although the OS execution does have impact on a superscalar.

2. Apache spends most of its time in the OS kernel, executing file system and networking operations.

3. The Apache OS-intensive workload is very stressful to a processor, causing significant increases in cache miss rates compared to SPECInt.

4. From our detailed analysis of conflict misses, there is significant interference between kernel threads on an SMT, because SMT can execute instructions from multiple kernel threads simultaneously. On the other hand, there are opportunities for benefiting from cooperative sharing, as we showed in our analysis of interthread prefetching.

5. Overall, operating system code causes poor instruction throughput on a superscalar. SMT's latency tolerance is able to compensate for many of the demands of operating system code. When executing Apache, SMT achieves a 4-fold improvement in throughput over the superscalar, the highest relative gain of any SMT workload to date.

Finally, we showed that it is relatively straightforward to modify an SMP-aware operating system to execute on a simultaneous multithreaded processor. In the future, we intend to experiment with OS structure in order to optimize the OS for the special features of SMT.

SMT processors are likely to be commonplace for a good while. SMT provides an effective way to extract more throughput from a processor without incurring a high implementation overhead. With Intel now producing SMT processors as standard and IBM starting to produce their SMT Power5 processor, the use of SMT will become more widespread. It is likely that future SMT processor will support more threads than current implementations because the overhead of adding

the extra state to the processor is fairly low. However, although early SMT research described processors with up to 8 threads and hinted at even larger numbers, practical considerations such as the size and complexity of the register file limit the scalability of SMT. Multicore processors, which have now been available for a few years, with a shared level 2 cache are likely to have their scalability limited by the connection and bandwidth between the cores and the cache. Both IBM and Sun Microsystems have announced processors which combine multithreading and multiple cores. It is likely that this combination will become more common in the future.

REFERENCES

- C.Zilles, J. Emer, and G. Sohi. The use of multithreading for exception handling. In *32nd Annual International Symposium on Microarchitecture*, November 1999.
- D.M. Tullsen and J. A. Brown. *Handling Long-latency Loads in a Simultaneous Multithreading Processor*. In Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO-34), pages 318. 327. IEEE Computer Society, December 2001. (p 51)
- D.M. Tullsen, S. J. Eggers, and H. M. Levy. *Simultaneous Multithreading: Maximizing On-Chip Parallelism*. In Proceedings of the 22th International Symposium on Computer Architecture (ISCA '95), pages 392.403. IEEE Computer Society, June 1995. (pp 20, 37)
- R.Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous subordinate microthreading (SSMT). In *26th Annual International Symposium on Computer Architecture*, May 1999.
- S.J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo R. L. Stamm, and D. M. Tullsen. *Simultaneous Multithreading: A Platform for Next-Generation Processors*. IEEE Micro, 17(5):12.19, October 1997. (pp 20, 37)
- S.Parekh, S. Eggers, and H. Levy. Thread-sensitive scheduling for smt processors. Technical report, Department of Computer Science & Engineering, University of Washington, 2000.
- T.Ungerer, B. Robi_c, and J. _ Silc. *A Survey of Processors with Explicit Multithreading*. ACM Computing Surveys, 35(1):29.63, March 2003. (p 20)
- U.Sigmund and T. Ungerer. Memory hierarchy studies of multimedia-enhanced simultaneous multithreaded processors for MPEC-2 video decompression. In *Workshop on Multi-Threaded Execution, Architecture and Compilation*, January 2000.
- Y.Hu, A. Nanda, and Q. Yang. Measurement, analysis and performance improvement of the apache web server. In *Proceedings of the 18th International Performance, Computing and Communications Conference*, February 1999.