

Exploring Simultaneous Multithreading

John Hall
CS 551

Abstract

This paper investigates simultaneous multithreading (SMT). This technique allows a processor to choose instructions from several streams at once. This paper introduces SMT in detail and shows how it can be further optimized. It compares it to multiprocessor systems, another popular technique available. We find SMT the clear winner for small-scale systems. This paper concludes by looking at some recent work and future work regarding SMT.

Introduction

Simultaneous multithreading (SMT) is a processor architecture proposed in 1996. It allows the hardware to execute multiple threads of execution at the same time. This allows the processor to dynamically choose the best instruction to execute from the pool of available instructions. The processor can make this choice in such a way that will best utilize the processor resources. For example, if one thread is stalled due to a memory latency, the other threads can continue to do meaningful work. This allows a processor to much more efficiently allocate its resources. There are even models that reduce the complexity of processor design by identifying and eliminating data forwarding and dependency issues at the dispatch phase. SMT allows a processor to execute instructions more efficiently and therefore can dispatch more instructions in a given time. While the performance of individual threads will be decreased, the overall performance of the system can be dramatically improved.

SMT is not a perfect. There are some negative impacts of adding SMT to a system. The largest of these is that multiple threads do not have the same degree of locality as a single thread. This makes it more difficult for the cache to maintain enough context for all threads. The number of registers required to support many contexts can also be a bottleneck. If the memory system was a bottleneck before adding SMT, SMT will probably make it worse. Finally, the dispatch unit is likely to become a bottleneck. This unit must manage between the contexts of all threads. If this unit is not fast enough to keep up, the whole system will slow down.

Architecture

The basic idea of SMT is that by providing multiple threads of execution, the processor can mix the instructions in such a way as to maximize the resource usage of the processor. This hides the latencies encountered by a given thread by allowing the processor to continue on other threads. However, it does more than just hide simple latencies. SMT allows the processor to balance normal fluctuations in resource usage. As a result, it allows thread that wish to use a popular resource to continue, but slowly, while threads that wish to use less popular resources will be dispatched more quickly. SMT allows the processor to more efficiently grant access to its resources among fluctuating demands. While SMT sounds like a major architectural upheaval, its strength lies in how easily this architecture can be added to an existing superscalar processor.

There are only several changes needed to update an existing well-designed processor to use SMT. The primary difference is that the system will need one register set for each thread. To identify which group of registers a given register is from it will need to have some tag bits added. This can be handled nicely by existing register renaming techniques. We also need to get the instructions streams to the the processor. As a result, the fetch unit will need to fetch from multiple threads. The only other component needed is a component to choose which instructions from the available instructions are dispatched. It is in this choice that the benefits of SMT are revealed.

The utilization of the processor depends heavily on how well we decide which instructions to

execute. However, the utilization of the processor is not the only consideration. The processor is measured by how much useful work it does. For example, if the processor is busy executing instructions that turn out to be on a mispredicted path of a branch, then the work is wasted. Just trying to keep the processor busy, for example, by using a round robin scheduler, will result in only a minor performance enhancement. SMT can also try to choose instructions that are less likely to stall. Despite lost cycles due to mispredicted branches and other poor choices of instructions, the first SMT simulation found a significant performance increase of 84%. [Tullsen, et al, 96] When compared to how difficult it's been for processors to improve their IPCs using previous techniques, this is quite a breakthrough.

SMT can deliver even more performance if we pick better instructions. The group then looked at some simple things they could check to determine a method of choosing the best instructions. The most effective method they found was simply to count the number of instructions currently operating through the system from each queue. [Tullsen, et al, 96] The processor would then dispatch instructions from the available threads with the smallest number. This method forces an instruction scheme that is well mixed from the available threads. It also allows threads that are effectively moving instructions through execution units to dispatch more instructions. Combined with a scheme to minimize the impact of an instruction cache miss, their SMT processor was improved by 37%. Combined with the earlier increase of 84%, their SMT processor executes two and a half times as many useful instructions per cycle as the original superscalar processor. These performance improvements are confirmed by other results. For example, [Lo, et al. 98] found that the improvements were actually a little greater in a disk-intensive database environment.

The Tera computer [Alverson, et al. 90], now known as the Cray MTA, uses fine-grained multithreading. This system deals with dependencies in an interesting way that could be extended to an SMT system. This system doesn't worry about dependencies. In fact, there is no cache. Each processor cycles between up to 128 different threads. By the time the next instruction gets processor time, the memory system has had up to 128 cycles to get the memory item to the processor. A system using SMT would be even more powerful. Such a system, hypothetically call it the Cray SMT, would allow the processor to operate on any combination of the instructions that have received their inputs from memory. This would allow the system to improve overall performance by favoring the threads using memory more efficiently. Each instruction includes a memory access, an arithmetic operation, and a control operation. SMT could schedule the available instructions to keep all three of these units busy, even when there was a noop in an instruction slot. If there tend to be more noops in one slot than others, or one of these operations could be made fast enough to handle multiple instructions per cycle, then the performance could be improved even more by adding an additional bottleneck resource. Such optimizations are not possible using other forms of fine-grained multithreading.

SMT has several advantages over multiprocessing. These can be simplified to the fact that a small incremental amount of hardware can result in a large performance gain [Tullsen, et al, 95]. With multiprocessing, the performance gain is not as straightforward. For simple problems that can be expressed in a completely parallel way with no shared resources, the performance increases

roughly linearly with the number of processors. The incremental performance of SMT falls dramatically as additional threads are added. [Hammond, et al. 97] claims that the abundance of transistors will drive to putting multiple processors on a chip. They have two arguments along these lines. The authors point out that it is much easier to design a single superscalar processor and duplicate it than it is to design a single, more complex, superscalar processor. They also point to the fact that communication times between components (interconnect delays) are not decreasing as quickly.

The design costs of an SMT processor are higher than for a chip multiprocessor (CMP) or the equivalent performance. However, the cost of the production chip and the failure rate are based on the area of the unit. The push to deliver faster performing systems will initially result in a CMP solution. However, after the first generation CMP, market forces will push to decrease the cost of that performance. A company that added SMT to a processor will be able to decrease the per unit cost because the costs will be lower. The push towards lower power processors will also favor the SMT processor. However, because of the limit to the number of threads on a SMT processor, eventually there will be CMPs made up of SMT processors. The same cycle will repeat itself again with these systems and the next great architectural breakthrough.

The other argument made was that interconnect delays are not decreasing as the number of transistors. This argument is based on the assumption that the communication paths in a SMT processor are significantly more complex than a superscalar without SMT. The dispatch and completion unit is not as large a portion of the complexity of a processor as the author would have us believe. [Jaffe, 01] shows a picture of a PowerPC™ 620 processor with the various components labeled. As you can see, even if the complexity and size of the dispatch and completion unit were doubled, the overall complexity of the processor would seem to increase by less than 20%.

SMT is a clear winner over CMP and similarly over traditional multiprocessors. A significantly smaller processor with significantly less overall complexity can generate significantly faster performance while using less power. Instead of duplicating resources, SMT allows a processor to make better use of its existing resources.

Most of the current work regarding SMT involves making use of spare threads to improve the operation of the existing threads. Some examples include fault tolerance, memory prediction, and branch prediction. There appear to still be many possibilities for uses of idle threads. The fault tolerance version runs additional versions of the executing threads on the idle threads. This allows failures to be detected and recovered from [Reinhardt, et al. 00]. The memory prediction scheme is based on the discovery that pre-fetching memory is often difficult. In order to predict what addresses a program will use may require running the program. If an SMT processor has idle threads, it could run the instructions from another thread to generate memory pre-fetches before the original thread needs the values [Luk, 01]. Finally, the branch prediction scheme allows less likely branch paths to run on idle threads. If the original prediction was incorrect, the processor can switch the threads instead of just flushing the incorrect path [Wallace, et al. 98]. These

techniques require a priority scheme so that the threads doing productive work are not slowed by the activities of these additional threads.

Conclusion

SMT is an exciting architectural breakthrough in computer architecture. By giving the processor the flexibility to choose instructions from multiple threads, the processor can hide latencies and maximize execution resource usage. SMT is clearly the best available way to extract additional performance for future architectures.

Annotated Bibliography

[Alverson, et al. 90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, Burton Smith. *The Tera Computer System* (1990).
<http://citeseer.nj.nec.com/alverson90tera.html>.

The Tera MTA computer in detail. (Handout given in class)

[Gunther, 93] Bernard K. Gunther. *Superscalar Performance in a Multithreaded Microprocessor* (1993). <http://citeseer.nj.nec.com/gunther93superscalar.html>.

Some background on basic multithreading. Its kind of surprising the SMT didn't appear sooner.

[Hammond, et al. 97] Lance Hammond, Basem A. Nayfeh, Kunle Olukotun. *A Single-Chip Multiprocessor* (1997). <http://citeseer.nj.nec.com/hammond97singlechip.html>.

This paper claims that multiple processors offer the best potential for the use of those billion transistors. This seems a blow in the face for SMT. I don't quite believe their argument. They are basically claiming that the "interchanges" between processor components will be the limiting factor. This article gives me an opportunity to show why conventional wisdom isn't quite right. I believe they have the right idea, but in the wrong place. They contend that the design cost and so forth is too high for SMT. I think they neglect that the high end market is willing to pay for that better performance.

[Hennessy, et al. 03] John L. Hennessy and David A. Patterson. *Computer Architecture-A Quantitative Approach*. Third Edition. Morgan Kaufmann Publishers, CA.

Includes a good introduction to SMT. Includes some performance information as well as an assessment of SMT compared to other methods. Required introductory reading.

[Jaffe, 01] Matt S. Jaffe. *Chips, Wafers, Dies, Masks, and Photolithography* (2001).
http://backoff.pr.erau.edu/jaffem/classes/cs470/cs470_supplement_1.htm

This is a course web site for a hardware at Embry-Riddle. (The school my brother went to.) I included this reference for the picture of the PowerPC 620 layout. (I couldn't find it on Motorola or IBM's pages.) I think it is obvious to see from this that the dispatch and completion unit as well as the registers are not a large portion of the overall layout.

[Laudon, et al. 94] James Laudon, Anoop Gupta, Mark Horowitz. *Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations* (1994).
<http://citeseer.nj.nec.com/laudon94interleaving.html>.

Fine-grained multithreading. This was right before SMT became well known. This basically

shows that fine-grained is better than course-grained.

[Lo, et al. 97] Jack L. Lo, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, Dean M. Tullsen. *Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading* (1997). <http://citeseer.nj.nec.com/lo97converting.html>.

Explores the gains of SMT in more detail. In particular, thread-level parallelism and ILP are interchangeable. This is much better than just exploiting ILP as on a typical processor. It is also better than multiple processors to exploit thread-level parallelism.

[Lo, et al. 98] Jack L. Lo, Luiz Andre Barroso, Susan Eggers, Kourosh Gharachorloo, Henry Levy, Sujay Parekh. *An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors* (1998). <http://citeseer.nj.nec.com/lo98analysis.html>.

Does some research to determine how well SMT can help real-world data intensive applications. It finds that indeed SMT works well in these environments. In fact, they find the same gains (actually a little more) than [Tullsen, et al. 96].

[Luk, 01] Chi-Keung Luk. *Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors* (2001). <http://citeseer.nj.nec.com/luk01tolerating.html>.

This is fascinating because it extends SMT by asking what can be done with idle threads. The idea is we can use spare threads to try to improve our guessing of upcoming memory accesses.

[Nemirovsky, 95] Mario Nemirovsky, Wayne Yamamoto, *Increasing Superscalar Performance Through Multistreaming*, 3rd Int. Symp. on Parallel Arch. and Compiler Techniques, June 1995. Unable to locate online.

As you mentioned, this appears to be the first description of SMT. I suspect this paper was not fully noticed because their simulation model is somewhat simplistic. I suspect that combined with the large performance increases, most people only read as far as the model before they discounted the paper. I suspect a similar thing happened when [Tullsen, et al., 95] was released. In this paper they, quite rightly, only promised the potential. I think it was the second paper [Tullsen, et al., 96] that was their claim to fame.

[Nemirovsky, 98] Mario Nemirovsky, Wayne Yamamoto. *Quantitative study of data caches on a multistreamed architecture* (1998). <http://citeseer.nj.nec.com/nemirovsky98quantitative.html>.

It seems that the reason Nemirovsky didn't get the credit for SMT is that this paper was late. The Tullsen papers were the first to make SMT reasonable.

[Reinhardt, et al. 00] Steven K. Reinhardt, Shubendu S. Mukherjee. *Transient Fault Detection*

via *Simultaneous Multithreading* (2000). <http://citeseer.nj.nec.com/reinhardt00transient.html>.

Another use for spare threads. Could we run an identical copy of a program on a spare thread and compare the output. Then we know if a hardware failure occurred. I believe this has a fairly large cost and there are better ways to achieve this.

[Stork. 00] Christian Stork. *Exploring the Tera MTA by Example* (2000). <http://citeseer.nj.nec.com/stork00exploring.html>.

Some more on the Tera example you mentioned in class. This paper looks at the other side of multithreading. In particular how well it has actually worked. While the paper is specific to the Tera MTA, it is still helpful in understanding how such parallelism can be used and the issues. (Which are slightly different than programming for multiple processors.)

[Tullsen, et al. 95] Dean Tullsen, Susan J. Eggers, Henry M. Levy. *Simultaneous Multithreading: Maximizing On-Chip Parallelism* (1995). <http://citeseer.nj.nec.com/tullsen95simultaneous.html>.

This paper introduces the term SMT. It shows the greater potential of SMT over using the space for multiple processors.

[Tullsen, et al. 96] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy Jack L. Lo, Rebecca L. Stamm. *Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor* (1996). <http://citeseer.nj.nec.com/tullsen96exploiting.html>.

Shows that SMT can be (relatively easily) added to a superscaler processor. Shows the large potential for performance increases and discusses possible optimizations. Note that this model allows the single thread case to run with only a minimal performance cost.

[Wallace, et al. 98] Steven Wallace, Brad Calder, Dean M. Tullsen. *Threaded Multiple Path Execution* (1998). <http://citeseer.nj.nec.com/wallace98threaded.html>.

Use spare contexts to execute unlikely paths from branch prediction. This does not have the same benefit as SMT in general.

[Warnakulasuriya] Sugath Warnakulasuriya. *The Use of Multithreaded Processors in DASH* (Unknown). <http://citeseer.nj.nec.com/14060.html>.

This paper has a good overview of DASH. However, more interesting are the portions analyzing the benefits and drawbacks to adding such a process. It appears to have been written fairly early in DASH's lifetime.