Simultaneous Multithreading: a Platform for Next Generation Processors

Paulo Alexandre Vilarinho Assis

Departamento de Informática, Universidade do Minho
4710 – 057 Braga, Portugal
paulo.assis@bragatel.pt

Abstract. To achieve high performance, contemporary processors rely mainly on two forms of parallelism: instruction level parallelism (ILP) and thread level parallelism (TLP). Unfortunately, both parallel processing styles statically partition processors resources. Simultaneous Multithreading (SMT) explores parallel processing on an alternative architecture. From superscalars it inherits the ability to issue multiple instructions each cycle, and like multithreaded processors it can execute several programs (or threads) at once. The result is a processor that can issue multiple instructions from multiple threads each cycle, capable of adapting to dynamically changing levels of ILP and TLP in a program.

1 Introduction

To achieve high performance contemporary processors rely on two forms of parallelism: instruction level parallelism (ILP) and thread level parallelism (TLP).
ILP and TLP are fundamentally identical: they both identify independent instructions that can execute in parallel and therefore can utilize parallel hardware.
Superscalar processors exploit ILP by executing multiple instructions from a single program on a single cycle. Multiprocessors exploit TLP by executing different threads in parallel on different processors.

Unfortunately, neither parallel processing style is capable of adapting to dynamically changing levels of ILP and TLP, because the hardware enforces the distinction between the two types of parallelism.
TLP can only be exploited by adding more processors, and ILP can only be exploited by adding more resources on each processor.

This paper tries to show a new approach to processors architecture to overcome the lack of flexibility to adapt to fluctuations in both forms of parallelism. In the paper is gone a be shown the new concept of architecture, why it evolved that way and what are the main differences between SMT, ILP and TLP, pointing SMT key advantages contributing to the rising of utilization of functional units and by doing so, boosting processors performance.

2 How SMT Works

Current microprocessors employ various techniques to increase parallelism and processor utilization; however, each technique has its limits. Superscalars issue up to four instructions in a cycle from a single thread. Multiple instruction issue has the potential to increase performance, but is ultimately limited by instructions dependencies and long latency operations within the single execution thread. The effects of these are shown as horizontal

waste and vertical waste. Horizontal waste occurs when some, but not all, of the issue slots in a cycle can be used. It generally occurs because of poor instruction level parallelism. Vertical waste occurs when a cycle goes totally unused. This can be caused by a long latency instruction (such as memory access) that inhibits further instruction issue. Multithreaded architectures, on the other hand employ multiple threads with fast context switch between threads. Traditional multithreading hides memory and functional units latency, attacking vertical waste.

Simultaneous multithreading provides a better chance for the highest hardware utilization, since it uses, both ILP and TLP in the same cycle. Each cycle an SMT processor selects instructions for execution from all threads. If one thread has a high level of ILP, that parallelism can be satisfied; If multiple threads each have low levels of ILP, they can be executed together to compensate for the low ILP in each. Consequently, simultaneous multithreading attacks both horizontal and vertical waste.

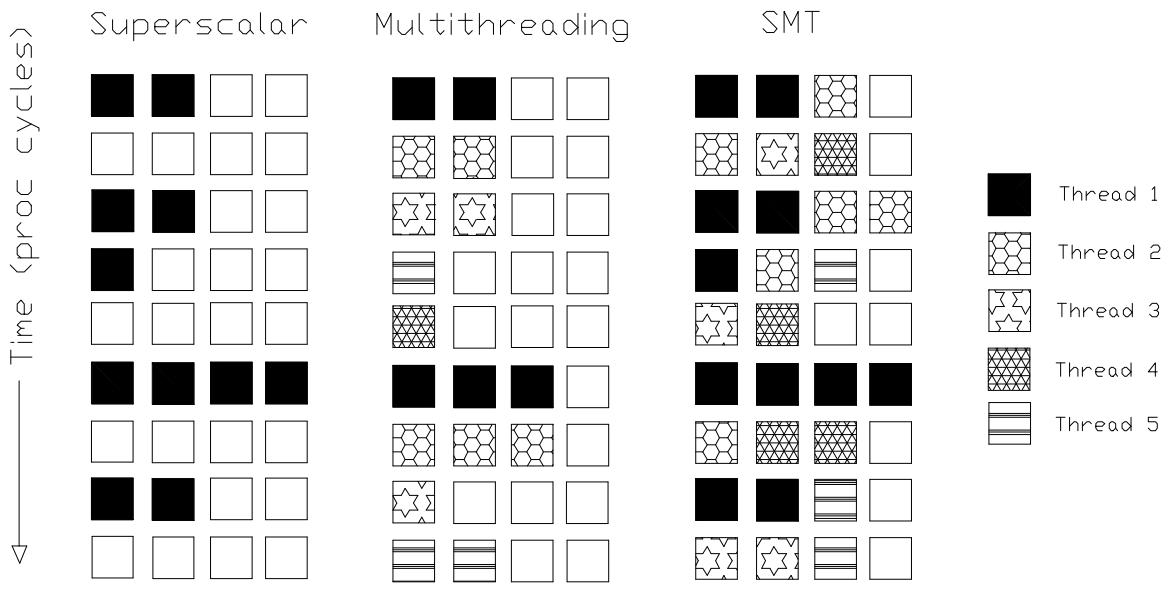

Figure 1: Comparison of issue slot (functional unit) partitioning in various architectures

3 Instruction Fetching

In a conventional processor, the instruction unit is responsible for fetching blocks of instructions from a single thread into the execution unit. The main performance issues revolve around maximizing the number of useful instructions that can be fetched, minimizing branch mispredictions, and fetching independent instructions quickly enough to keep functional units busy.

In an SMT processor, the fetch unit is shared among the available multiple threads. The fetch unit can fetch instructions from multiple threads simultaneously increasing the fetch bandwidth utilization. It can be selective about which thread or threads to fetch instructions from. Since all threads cannot provide equally useful instructions in a particular cycle, fetching from thread(s) will provide the best instructions will benefit the SMT processor.

An SMT processor places additional stress on the fetch unit. First, it is responsible for fetching instructions from up to eight different threads, and second, it has more difficult to

keep up with a efficient dynamic scheduler, witch can issue more instructions each cycle, because it takes them from multiple threads. Consequently, the fetch unit is SMT`s performance bottleneck.

2.1 "2.8" Fetch

The "2.8" Fetch is a fetching scheme that splits the fetch over multiple threads and is considered the best-fetching scheme, capable of achieve higher maximum throughput.
In every cycle, it selects two different threads from a group of threads that have not incurred instruction cache misses earlier and reads eight instruction block for each thread.

It takes instructions from the first thread until it comes across a branch instruction or the end of cache line and the rest are filled in from the second thread to make a total of eight. This scheme involves only minor hardware additions.

Two hit/miss calculations per cycle are required. It has two cache output buses, each eight instructions wide. In addition, logic to select and combine instructions is necessary and additional pipe stage might be needed.

2.2 ITAG

A fetch unit is blocked when the IQ is full or there is a miss in the instruction cache. ITAG scheme is used to prevent the blocking of the fetch unit, when a thread is selected for fetching but experiences a cache miss.

Only non-missing threads are chosen for fetch and at the same time cache miss accesses are started. This adds an additional stage to the front of the pipeline, increasing misfetch and mispredict penalties.

It is not an effective scheme when there are few threads.

2.3 Icount Heuristic

Icount heuristic is used to select the best thread in an SMT processor. It gives highest priority to the threads that have lowest instructions in the decode, renaming and queue pipeline stages. It improves the performance in several ways.

It uses a counter to keep track of instruction count in each thread. Therefore, it requires a small amount additional logic to increment (decrement) counters in each thread when instructions enter the decode stage and to select two smallest counter value threads.

The improved performance is because it dynamically biases toward threads that will use processor resources efficiently.

3 Register File and Pipeline

In simultaneous multithreading (as in a superscalar processor), each thread can address 32 architectural integer (and floating point) registers. The register-renaming mechanism maps these architectural registers onto a hardware register file whose size is determined by the number of architectural registers in all thread contexts, plus a set of additional renaming registers. The larger SMT register file requires a longer access time; to avoid increasing the processor cycle time, the SMT pipeline was extended two stages to allow two-cycle regis-

ter reads and two-cycle writes. On the first SMT register read cycle, data is read the register file into a buffer. Then in the next cycle data is sent to a functional unit for execution: Writes to the register file behaviour in a similar manner, also using an extra pipeline stage.

The two-stage register access has several ramifications on the architecture. For example, the extra stage between instruction fetch and execute increases the branch misprediction penalty by one cycle. The two extra stages between register renaming and instruction commit increase the minimum time that an executing instruction can hold a hardware register; this, in turn, increases the pressure on the renaming registers.

Finally, the extra stage needed to write back results to the register file requires an extra level of bypass logic.

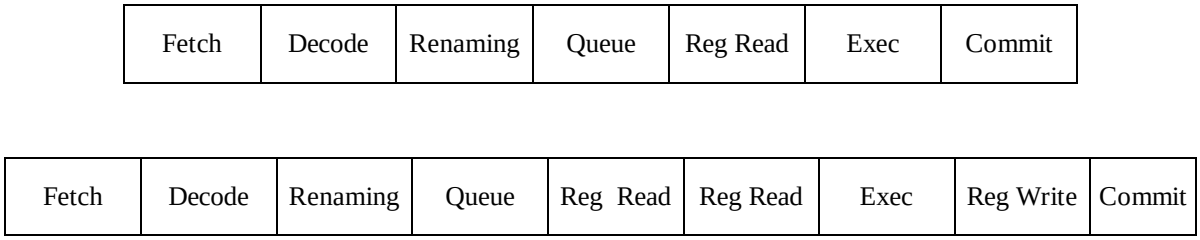

Figure 2: Comparison of the pipelines for a conventional superscalar processor (top) and SMT (bottom).

4 Latencies and Bandwidth

High latencies can adversely affect the rate at processors execute instructions, and thus, performance. Caches greatly alleviate this problem, and caches have, arguably, a larger role to play in multiprocessor system than they do in uniprocessor system.

Threads on an SMT processor share the same cache hierarchy, so their working sets may introduce inter-thread conflict misses. We identify inter-thread misses as the misses that would have been hits, if each thread had its own private 32KB L1 D cache. As the number of threads increases, the number of inter-thread conflict misses also rises. The primary concern, however, is not the impact on the hit rate, but the impact on overall performance. There are two reasons why inter-thread cache interference is not a significant problem for our workload. First, the additional inter-thread conflict misses in the direct-mapped L1 cache are almost entirely covered by the 4-way set associative cache. The fully pipelined L2 cache has a relatively low latency (6 additional cycles) and high bandwidth, so average memory access time (AMAT) increases only slightly. When increasing the number of threads from 1 to 8, the cache miss component of AMAT increases by less than 1.5 cycles on average, indicating the small effect of inter-thread conflict misses. Second, out of order execution, write buffering, and the use of multiple threads allow SMT to hide the small increases in additional memory latency, and large speedups can be attained.

5 Difference between Superscalar, Multiprocessors and SMT

5.1 SMT vs. the Superscalar

Superscalar processor executes a single program or thread. It attempts to issue multiple instructions each cycle of a program. Due to low instruction-level parallelism, it may not find sufficient number of instructions to issue, causing horizontal and vertical wastes.

SMT processor fetches instructions from all threads for execution each cycle. It exploits both instruction-level parallelism and thread-level parallelism. It can execute only one thread if it has highest instruction-level parallelism or it can execute multithreads if each thread has a low instruction-level parallelism. Thus, it recovers issue slots lost to both horizontal and vertical wastes.

5.2 SMT vs. the Multiprocessors

The fixed partitioning of the multiprocessors hardware resources prevents them from responding well to changes in instruction and thread level parallelism. Processors can be idle when thread level parallelism is insufficient; and processors have trouble exploiting large amounts of instruction level parallelism in the unrolled loops of individual threads. An SMT processor, on the other hand, dynamically partitions its resources among threads, and therefore can respond well to variations in both types of parallelism, exploiting them interchangeably. When only one thread is executing, (almost) all machine resources can be dedicated to it; and additional threads can compensate for a lack of instruction level parallelism in any single thread.

On the organizational level, multiprocessors and simultaneous multithreading processors are very similar. Both have multiple register sets, multiple functional units, and high issue bandwidth on a single chip. The key difference is in the way those resources are partitioned and scheduled. The multiprocessor statically partitions resources, devoting a fixed number of resources to each thread. The simultaneous multithreading processor allows the partitioning to change every cycle. It should be obvious that the job of scheduling is more complex on an SMT processor than a multiprocessor. But, in other areas, the simultaneous multithreading model requires fewer resources than the multiprocessor to achieve the desired level of performance.

6 Conclusions

Simultaneous multithreading allows compilers and programmers to focus on extracting whatever parallelism exists, by treating instruction and thread level parallelism equally. ILP and TLP are fundamentally identical; they both represent independent instructions that can be used to increase processor utilization and improve performance. SMT has the flexibility to use both forms of parallelism interchangeably, because threads can share resources dynamically. Rather than adding more resources to further improve performance, existing resources are used more effectively. By using more hardware contexts, SMT can take advantage of TLP to expose more parallelism.

Even though these parallel threads have greater potential for interference because of similar resource usage patterns (including memory references and demands for renaming registers and functional units), simultaneous multithreading has the ability to compensate for these potential conflicts. It is reasonable to think that inter-thread cache interference, bank contention, and branch prediction interference on an SMT processor has only minimal effects on performance.

The bottom line is that simultaneous multithreading makes better utilization of on chip resources to run parallel applications effectively.

References

[1] 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 www.cs.washington.edu/research/smt/papers /tlp2ilp.final.pdf

[2] Dean M. Tullsen, Susan J. Eggers and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism www.cs.utexas.edu/users/dburger/teaching/fall01/ cs382m/papers/ simultaneous-multithreading-maximizing-on.pdf

[3] Nihar R. Mahapatra and Balakrisna Venkatrao. The processor-Memory bottleneck: Problems and Solutions www.acm.org/crossroads/xrds5-3/pmgap.html

[4] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, Dean M. Tullsen. Simultaneous Multithreading: A Foundation for Next Generation Processors www.eecg.toronto.edu/~brewste/java/jit/jit-ieee-micro.ps

[5] Craig B. Zilles, Joel S. Emer and Gurindar S. Hohi. The Use of Multithreading For Exception Handling www.cs.wisc.edu/~zilles/papers/except-thrd.micro.pdf

[6] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, Dean M. Tullsen. Simultaneous Multithreading: A Platform for Next-generation Processors 1997 IEEE Micro