Simultaneous Multithreading

Nicolae-Andrei Vasile

Faculty of Computer Science and Automatic Control

POLITEHNICA University of Bucharest

Bucharest, Romania

Abstract – Simultaneous multithreading is a technique that permits multiple independent threads to issue multiple instructions each cycle. Simultaneous multithreading has the potential to achieve 4 times the throughput of a superscalar, and double that of fine-grain multithreading. Simultaneous multithreading is also an attractive alternative to single-chip multiprocessors; simultaneous multithreaded processors with a variety of organizations outperform corresponding conventional multiprocessors with similar execution resources. While simultaneous multithreading has excellent potential to increase processor utilization, it can add substantial complexity to the design.

I. INTRODUCTION

Simultaneous multithreading (SMT) is a technique that permits multiple independent threads to issue multiple instructions each cycle to a superscalar processor's functional units. SMT combines the multiple-instruction-issue features of modem superscalars with the latency-hiding ability of multithreaded architectures. Unlike conventional multithreaded architectures, which depend on fast context switching to share processor execution resources, all hardware contexts in an SMT processor are active simultaneously, competing each cycle for all available resources. This dynamic sharing of the functional units allows simultaneous multithreading to substantially increase throughput, attacking the two major impediments to processor utilization, long latencies and limited per-thread parallelism [1], [2].

Simultaneous multithreading is a processor design that can exploit all types of parallelism well, because it consumes both thread-level and instruction-level parallelism. In SMT processors, thread-level parallelism can come from either multithreaded, parallel programs or individual, independent programs in a multiprogramming workload. Instruction-level parallelism comes from each single program or thread. Because it successfully (and simultaneously) exploits both types of parallelism, SMT processors use resources more efficiently, and both instruction throughput and speedups are greater.

Further, it combines hardware features of wide-issue superscalars and multithreaded processors. From superscalars, it inherits the ability to issue multiple instructions each cycle; and like multithreaded processors it contains hardware state for several programs (or threads). The result is a processor that can issue multiple instructions from multiple threads each cycle, achieving better performance for a variety of workloads. For a mix of independent programs (multiprogramming), the overall throughput of the machine is improved. Similarly, programs that are parallelizable, either by a compiler or a programmer, reap the same throughput benefits, resulting in program speedup. Finally, a single-threaded program that must execute alone will have all machine resources available to it and will maintain roughly the same level of performance as when executing on a single-threaded, wide issue processor [3].

Equal in importance to its performance benefits is the simplicity of SMT's design. Simultaneous multithreading adds minimal hardware complexity to, and, in fact, is a straightforward extension of, conventional

dynamically scheduled superscalars. Hardware designers can focus on building a fast, single threaded superscalar, and add SMT's multithread capability on top.

II. SIMULTANEOUS MULTITHREADING WORKFLOW

The difference between superscalar, multithreading, and simultaneous multithreading is pictured in Figure 1, which shows sample execution sequences for the three architectures. Each row represents the issue slots for a single execution cycle: a filled box indicates that the processor found an instruction to execute in that issue slot on that cycle; an empty box denotes an unused slot. The unused slots are characterized as horizontal or vertical waste. Horizontal waste occurs when some, but not all, of the issue slots in a cycle can be used. It typically occurs because of poor instruction level parallelism. Vertical waste occurs when a cycle goes completely unused. This can be caused by a long latency instruction (such as a memory access) that inhibits further instruction issue [3].

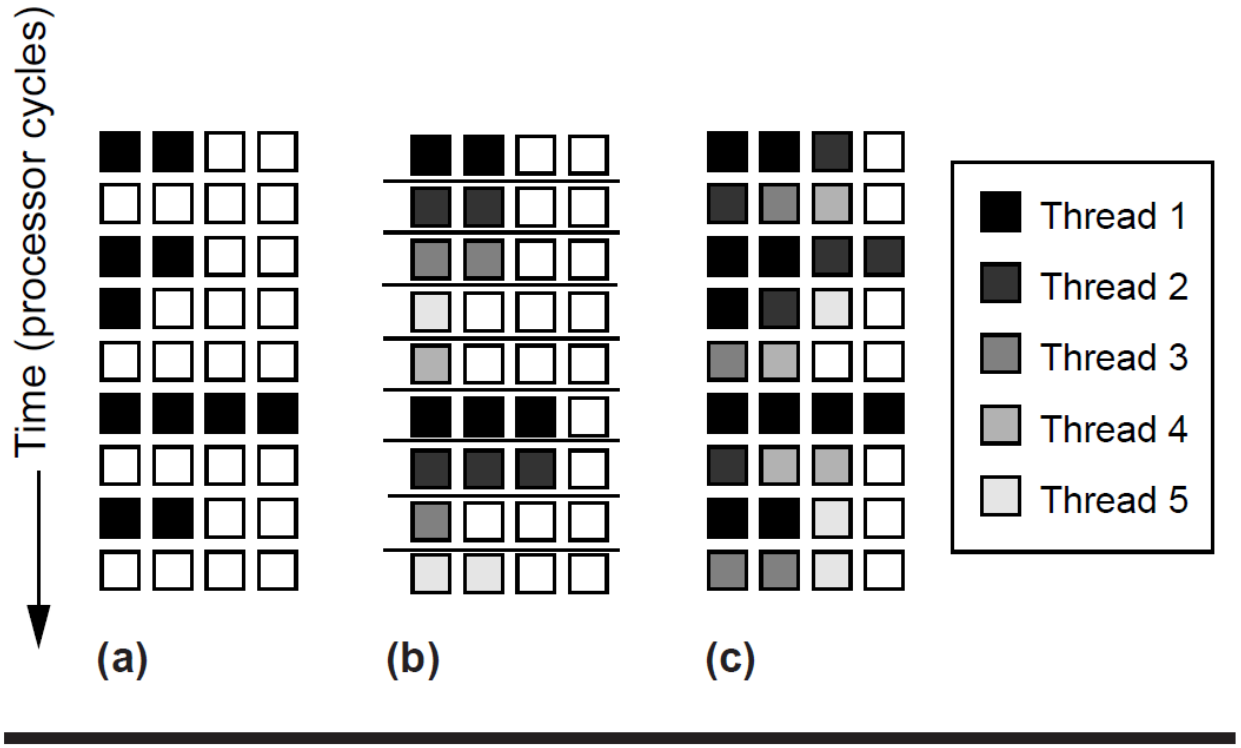

Figure 1. How architectures partition issue slots (functional units) [3]

Figure 1a shows a sequence from a conventional superscalar. As in all superscalars, it is executing a single program, or thread, from which it attempts to find multiple instructions to issue each cycle. When it cannot, the issue slots go unused, and it incurs both horizontal and vertical waste.

Figure 1b shows a sequence from a multithreaded architecture, such as the Tera. Multithreaded processors contain hardware state (a program counter and registers) for several threads. On any given cycle a processor executes instructions from one of the threads. On the next cycle, it switches to a different thread context and executes instructions from the new thread. As the figure shows, the primary advantage of multithreaded processors is that they better tolerate long latency operations, effectively eliminating vertical

waste. However, they cannot remove horizontal waste. Consequently, as instruction issue width continues to increase, multithreaded architectures will ultimately suffer the same fate as superscalars: they will be limited by the instruction-level parallelism in a single thread.

Figure 1c shows how each cycle an SMT processor selects instructions for execution from all threads. It exploits instruction-level parallelism by selecting instructions from any thread that can (potentially) issue. The processor then dynamically schedules machine resources among the instructions, providing the greatest chance for the highest hardware utilization. If one thread has high instruction-level parallelism, that parallelism can be satisfied; if multiple threads each have low instruction-level parallelism, they can be executed together to compensate. In this way, SMT can recover issue slots lost to both horizontal and vertical waste.

III. PERFORMANCE OVERVIEW

Figure 3 shows the performance of the various models as a function of the number of threads. The segments of each bar indicate the throughput component contributed by each thread. The bar-graphs show three interesting points in the multithreaded design space: fine-grained multithreading (only one thread per cycle, but that thread can use all issue slots), SM: Single Issue (many threads per cycle, but each can use only one issue slot), and SM: Full Simultaneous Issue (many threads per cycle, any thread can potentially use any issue slot).

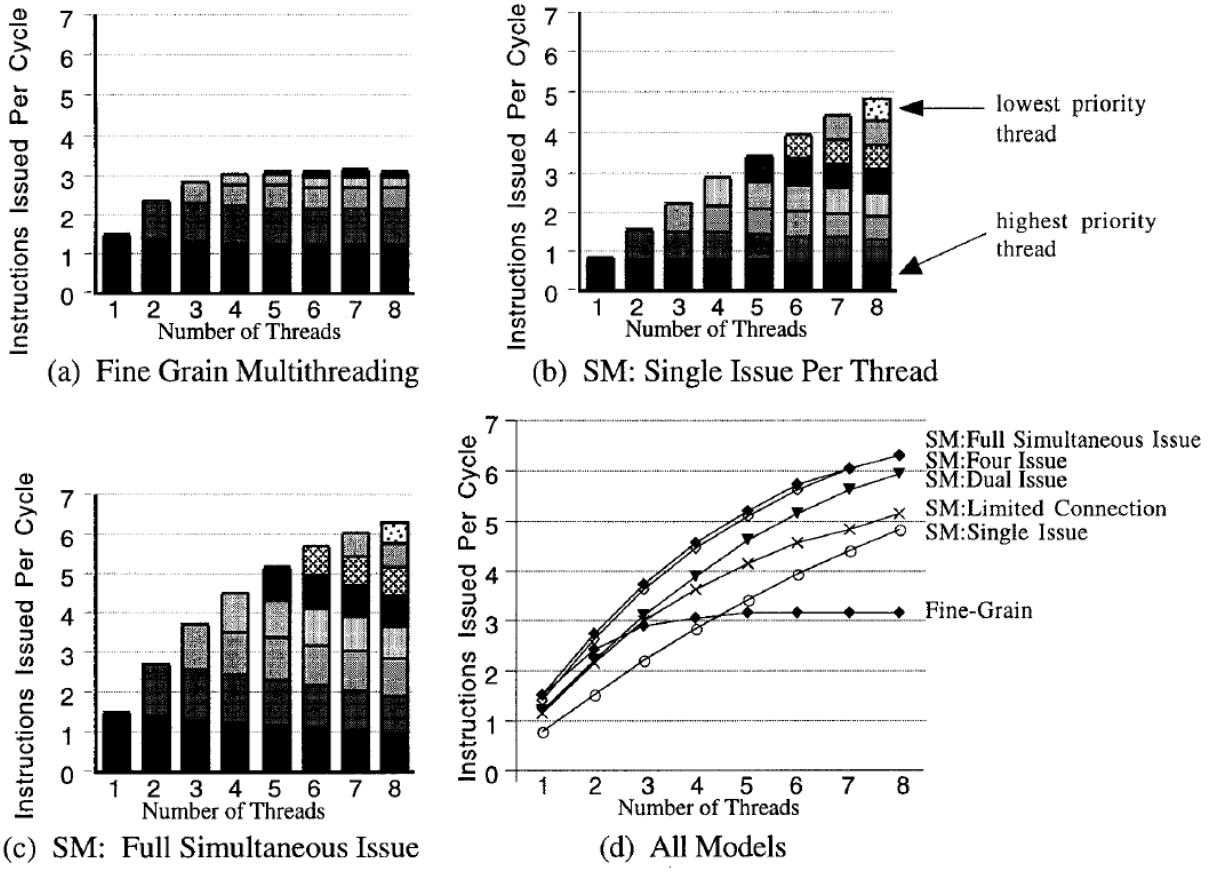

Figure 2. Instruction throughput as a function of the number of threads [1]

The fine-grain multithreaded architecture (Figure 3(a)) provides a maximum speedup (increase in instruction throughput) of only 2.1 over single-thread execution (from 1.5 IPC to 3.2). The graph shows that there is little advantage to adding more than four threads in this model. In fact, with four threads, the vertical waste has been reduced to less than 3%, which bounds any further gains beyond that point. This result is similar to previous studies for both coarse-grain and fine-grain multithreading on single-issue processors, which have concluded that multithreading is only beneficial for 2 to 5 threads. These limitations do not apply to simultaneous multithreading, however, because of its ability to exploit horizontal waste.

Figures 3b, 3c and 3d show the advantage of the simultaneous multithreading models, which achieve maximum speedups over single-thread superscalar execution ranging from 3.2 to 4.2, with an issue rate as high as 6.3 IPC. The speedups are calculated using the full simultaneous issue, 1-thread result to represent the single-thread superscalar.

With SM, it is not necessary for any single thread to be able to utilize the entire resources of the processor in order to get maximum or near-maximum performance. The four-issue model gets nearly the performance of the full simultaneous issue model, and even the dual-issue model is quite competitive, reaching 94% of full simultaneous issue at 8 threads. The limited connection model approaches full simultaneous issue more slowly due to its less flexible scheduling. Each of these models becomes increasingly competitive with full simultaneous issue as the ratio of threads to issue slots increases.

With the results shown in Figure 3(d), we see the possibility of trading the number of hardware contexts against hardware complexity in other areas. For example, if we wish to execute around four instructions per cycle, we can build a four-issue or full simultaneous machine with 3 to 4 hardware contexts, a dual-issue machine with 4 contexts, a limited connection machine with 5 contexts, or a single-issue machine with 6 contexts. Tera [3] is an extreme example of trading pipeline complexity for more contexts; it has no forwarding in its pipelines and no data caches, but supports 128 hardware contexts.

The increases in processor utilization area direct result of threads dynamically sharing processor resources that would otherwise remain idle much of the time; however, sharing also has negative effects. We see (in Figure 3(c)) the effect of competition for issue slots and functional units in the full simultaneous issue model, where the lowest priority thread (at 8 threads) runs at 55% of the speed of the highest priority thread. We can also observe the impact of sharing other system resources (caches, TLBs, branch prediction table); with full simultaneous issue, the highest priority thread, which is fairly immune to competition for issue slots and functional units, degrades significantly as more threads are added (a 35% slowdown at 8 threads). Competition for non-execution resources, then, plays nearly as significant a role in this performance region as the competition for execution resources.

Others have observed that caches are more strained by a multithreaded workload than a single-thread workload, due to a decrease in locality [21, 33, 1, 31]. Our data (not shown) pinpoints the exact areas where sharing degrades performance. Sharing the caches is the dominant effect, as the wasted issue cycles (from the perspective of the first thread) due to I cache misses grows from 170 at one thread to 14% at eight threads, while wasted cycles due to data cache misses grows from 12% to 18%. The data TLB waste also increases, from less than 170 to 6%. In the next section, we will investigate the cache problem. For the data TLB, we found that, with our workload, increasing the shared data TLB from 64 to 96 entries brings the wasted cycles (with 8 threads) down to 1%, while providing private TLBs of 24 entries reduces it to under 2Y0, regardless of the number of threads.

It is not necessary to have extremely large caches to achieve the speedups shown in this section. Our experiments with significantly smaller caches (not shown here) reveal that the size of the caches affects 1-thread and 8-thread results equally, making the total speedups relatively constant across a wide range of cache sizes. That is, while 8-thread execution results in lower hit rates than 1-thread execution, the relative effect of changing the cache size is the same for each.

In summary, results show that simultaneous multithreading surpasses limits on the performance attainable through either single-thread execution or fine-grain multithreading, when run on a wide superscalar. It has also been seen that simplified implementations of SM with limited per-thread capabilities can still attain high instruction

throughput. These improvements come without any significant tuning of the architecture for multithreaded execution; in fact, we have found that the instruction throughput of the various SM models is somewhat hampered by the sharing of the caches and TLBs. The next section investigates designs that are more resistant to the cache effects.

IV. SIMULTANEOUS MULTITHREADING VERSUS SINGLE-CHIP MULTIPROCESSING

As chip densities continue to rise, single-chip multiprocessors will provide an obvious means of achieving parallelism with the available real estate [1], [2], [3]. This section compares the performance of simultaneous multithreading to small-scale, single-chip multiprocessing (MP). On the organizational level, the two approaches are extremely 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 functional units to each thread; the SM processor allows the partitioning to change every cycle. Clearly, scheduling is more complex for an SM processor however, we will show that in other areas the SM model requires fewer resources, relative to multiprocessing, in order to achieve a desired level of performance.

For these experiments, “Simultaneous Multithreading: Maximizing On-Chip Parallelism” paper [1] tried to choose SM and MP configurations that are reasonably equivalent, although in several cases we biased in favor of the MP. For most of the comparisons all or most of the following are kept equal: the number of register sets (i.e., the number of threads for SM and the number of processors for MP), the total issue bandwidth, and the specific functional unit configuration. A consequence of the last item is that the functional unit configuration is often optimized for the multiprocessor and represents an inefficient configuration for simultaneous multithreading. All experiments use 8 KB private instruction and data caches (per thread for SM, per processor for MP), a 256 KB 4-way set-associative shared second-level cache, and a 2 MB direct-mapped third-level cache.

MPs with 1, 2, and 4 issues per cycle on each processor are evaluated. SM processors with 4 and 8 issues per cycle are evaluated; however we use the SM: Four Issue model (defined in Section 4.1) for all of our SM measurements (i.e., each thread is limited to four issues per cycle). Using this model minimizes some of the inherent complexity differences between the SM and MP architectures. For example, an SM: Four Issue processor is similar to a single-threaded processor with 4 issues per cycle in terms of both the number of ports on each register file and the amount of inter-instruction dependence checking. In each experiment it runs the same version of the benchmarks for both configurations (compiled for a 4-issue, 4 functional unit processor, which most closely matches the MP configuration) on both the MP and SM models; this typically favors the MP.

While in general we have tried to bias the tests in favor of the MP, the SM results may be optimistic in two aspects: the amount of time required to schedule instructions onto functional units, and the shared cache access time. The distance between the load/store units and the data cache can have a large impact on cache access time. The multiprocessor, with private caches and private load/store units, can minimize the distances between them. The SM processor cannot do so, even with private caches, because the load/store units are shared. However, two alternate configurations could eliminate this difference. Having eight load/store units (one private unit per thread, associated with a private cache) would still allow to match MP performance with fewer than half the total number of MP functional units (32 vs. 15). Or with 4 load/store units and 8 threads, it could statically share a single cache load/store combination among each set of 2 threads. Threads O and 1 might share one load/store unit, and all accesses through that load/store unit would go to the same cache, thus allowing us to minimize the distance between cache and load/store unit, while still allowing resource sharing.

Purpose of Test	Common Elements	Specific Configuration	Throughput (instructions/cycle)
Unlimited FUs: equal total issue bandwidth, equal number of register sets (processors or threads)	**Test A**: FUs = 32 Issue bw = 8 Reg sets = 8	SM: 8 thread, 8-issue	6.64
		MP: 8 1-issue procs	5.13
	Test B: FUs = 16 Issue bw = 4 Reg sets = 4	SM: 4 thread, 4-issue	3.40
		MP: 4 1-issue procs	2.77
	Test C: FUs = 16 Issue bw = 8 Reg sets = 4	SM: 4 thread, 8-issue	4.15
		MP: 4 2-issue procs	3.44
Unlimited FUs: Test A, but limit SM to 10 FUs	**Test D**: Issue bw = 8 Reg sets = 8	SM: 8 thread, 8 issue, 10 FU	6.36
		MP: 8 1-issue procs, 32 FU	5.13
Unequal Issue BW: MP has up to four times the total issue bandwidth	**Test E**: FUs = 32 Reg sets = 8	SM: 8 thread, 8-issue	6.64
		MP: 8 4-issue procs	6.35
	Test F: FUs = 16 Reg sets = 4	SM: 4 thread, 8-issue	4.15
		MP: 4 4-issue procs	3.72
FU Utilization: equal FUs, equal issue bw, unequal reg sets	**Test G**: FUs = 8 Issue BW = 8	SM: 8 thread, 8-issue	5.30
		MP: 2 4-issue procs	1.94

Figure 3. Various multiprocessor vs. simultaneous multithreading comparisons [1]

Figure 3 shows the results of our SM/MP comparison for various configurations. Tests A, B, and C compare the performance of the two schemes with an essentially unlimited number of functional units (FUS); i.e., there is a functional unit of each type available to every issue slot. The number of register sets and total issue bandwidth are constant for each experiment, e.g., in Test C, a 4 thread, 8-issue SM and a 4-processor, 2-issue-per-processor MP both have 4 register sets and issue up to 8 instructions per cycle. In these models, the ratio of functional units (and threads) to issue bandwidth is high, so both configurations should be able to utilize most of their issue bandwidth. Simultaneous multithreading, however, does so more effectively.

Test D repeats test A but limits the SM processor to a more reasonable configuration (the same 10 functional unit configuration used throughout this paper). This configuration outperforms the multiprocessor by nearly as much as test A, even though the SM configuration has 22 fewer functional units and requires fewer forwarding connections.

In tests E and F, the MP is allowed a much larger total issue bandwidth. In test E, each MP processor can issue 4 instructions per cycle for a total issue bandwidth of 32 across the 8 processors; each SM thread can also issue 4 instructions per cycle, but the 8 threads share only 8 issue slots. The results are similar despite the disparity in issue slots. In test F, the 4-thread, 8-issue SM slightly outperforms a 4-processor, 4-issue per processor MP, which has twice the total issue bandwidth. Simultaneous multithreading performs well in these tests, despite its handicap, because the MP is constrained with respect to which 4 instructions a single processor can issue in a single cycle.

Test G shows the greater ability of SM to utilize a fixed number of functional units. Here both SM and MP have 8 functional units and 8 issues per cycle. However, while the SM is allowed to have 8 contexts (8 register sets), the MP is limited to two processors (2 register sets), because each processor must have at least 1 of each of the 4 functional unit types. Simultaneous multithreading's ability to drive up the utilization of a fixed number of functional units through the addition of thread contexts achieves more than 2 ~ times the throughput.

These comparisons show that simultaneous multithreading outperforms single-chip multiprocessing in a variety of configurations because of the dynamic partitioning of functional units. More important, SM requires many fewer resources (functional units and instruction issue slots) to achieve a given performance level. For example, a

single 8-thread, 8-issue SM processor with 10 functional units is 24 times faster than the 8-processor, single-issue MP (Test D), which has identical issue bandwidth but requires 32 functional units; to equal the throughput of that 8-thread 8-issue SM, an MP system requires eight 4-issue processors (Test E), which consume 32 functional units and 32 issue slots per cycle.

V. CONCLUSIONS

This paper examined simultaneous multithreading, a technique that allows independent threads to issue instructions to multiple functional units in a single cycle. Simultaneous multithreading combines facilities available in both superscalar and multithreaded architectures.

The advantage of simultaneous multithreading, compared to the other approaches, is its ability to boost utilization by dynamically scheduling functional units among multiple threads. SM also increases hardware design flexibility; a simultaneous multithreaded architecture can tradeoff functional units, register sets, and issue bandwidth to achieve better performance, and can add resources in a fine-grained manner.

Simultaneous multithreading increases the complexity of instruction scheduling relative to superscalars, and causes shared resource contention, particularly in the memory subsystem. However, we have shown how simplified models of simultaneous multithreading reach nearly the performance of the most general SM model with complexity in key areas commensurate with that of current superscalars.

VI. REFERENCES

[1] D. M. Tullsen, S. J. Eggers and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings 22nd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, 1995.

[2] H. M. Levy, J. L. Lo, J. S. Emer, R. L. Stamm, S. J. Eggers and D. M. Tullsen, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," in *23rd Annual International Symposium on Computer Architecture*, Philadelphia, PA, USA, 1996.

[3] 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*, vol. 17, no. 5, pp. 12-19, 1997.