High-Performance Throughput Computing

Throughput computing, achieved through multithreading and multicore technology, can lead to performance improvements that are 10 to 30× those of conventional processors and systems. However, such systems should also offer good single-thread performance. Here, the authors show that hardware scouting increases the performance of an already robust core by up to 40 percent for commercial benchmarks.

Shailender Chaudhry
Paul Caprioli
Sherman Yip
Marc Tremblay
Sun Microsystems

Microprocessors took over the entire computer industry in the 1970s and 1980s, leading to the phrase "the attack of the killer micros" as a popular topic on the newsgroup comp.arch for many years. We believe that the disruption offered by throughput computing is similar in scope. Over the next several years, we expect *chip multithreading* (CMT) processors to take over all computing segments, from laptops and desktops to servers and supercomputers, effectively creating "the attack of throughput computing." Most microprocessor companies have now announced plans and/or products with multiple cores and/or multiple threads per core. Sun Microsystems has been developing CMT technology for almost a decade through the Microprocessor Architecture for Java Computing (MAJC, pronounced "magic") program[1,2] and a variety of SPARC processors.[3-7] IBM, through the Power4 and Power5;[8,9] and Intel, through hyperthreading, are shipping products in this space. AMD has announced plans to ship dual-core chips in late 2005. What is misunderstood, though, is how you can build a microprocessor that has several multithreaded cores that still delivers high-end single-thread performance. The combination of high-end single-thread performance and a high degree of multicore, multithreading (for example, tens of threads) is what we call *high performance throughput computing*, the topic of this article.

This article was based on a keynote speech by Marc Tremblay at the 2004 International Symposium on Computer Architecture in Munich, with some updated information.

A variety of papers have shown how workloads running on servers[10] and desktops[11] can greatly benefit from thread-level parallelism. It is no surprise that large symmetric multiprocessors (SMPs) were so successful in the 1990s in delivering very high throughput in terms of transactions per second, for instance, while running large, scalable, multithreaded applications. Successive generations of SMPs delivered better individual processors, better interconnects (certainly in terms of bandwidth, not necessarily in terms of latency) and better scalability. Gradually, applications and operating systems have matched the hardware scalability. A decade later, architects designing CMT processors face a dilemma: Many applications and some operating systems already scale to tens of threads. This leads to the question, "Should you populate a processor die with many very simple cores or should you use fewer but more powerful cores?" We simulated (and built, or are building) two very different cores and two different threading models and discuss how they benefit from multithreading and from growing the number of cores on a chip. We show that if multithreading is the prime feature and the rest of the core (that is, the rest of the pipeline and the memory subsystem) is architected around it, as opposed to simply adding threading to an existing core, these two different cores benefit greatly from multithreading (a 1.8 to 3.2× increase in throughput) and from multicores (3.1 to 3.7× increase). Server workloads dictate that high throughput be the primary goal. On the other hand, the impact of critical sections, Amdahl's law, response time requirements, and pipeline efficiency force us to try to design a high-performance single-thread pipeline although not at the expense of throughput.

Unfortunately as we discuss in a later section, two big levers that industry traditionally used—clock rate and traditional out-of-order execution—no longer work very well in the context of an aggressive 65-nm process and memory latencies now ranging in the hundreds of cycles.

Clearly the industry needs new techniques. This article describes *hardware scouting* in which the processor launches a hardware thread (invisible to software) which runs in front of the head thread. The goal here is to bring all interesting data and instructions (and control state) into the on-chip caches. Scouting heavily leverages some of the existing threaded hardware to boost single-thread performance. The control speculation accuracy, the scout's depth, the memory-level parallelism (MLP) impact, and the overall effect on performance and cache sizes form the core of this article. We will show that this microarchitecture technique will, for some configurations, improve performance by 40 percent on TPC-C, double the effective L2 cache size for SPECint2000, and make a 1-Mbyte L2 cache behave as a 64-Mbyte one for SPECfp2000.

Researchers have proposed several other techniques to reduce the impact of ever-increasing memory latencies, as the "Scaling the Memory Wall" sidebar explains.

Throughput computing

Systems designed for throughput computing emphasize the overall work performed over a fixed time period as opposed to focusing on a metric describing how fast a single core or a thread executes a benchmark. The work is the aggregate amount of computation performed by all functional units, all threads, all cores, all chips, all coprocessors (such as network and cryptography accelerators) and all the network interface cards in a system.

The scope of throughput computing is broad, especially if the microarchitecture and operating system collaborate to enable a thread to handle a full process. In this way, a 32-thread CMT can appear to be an SMP of 32 virtual cores, from the viewpoint of the operating system and applications. At one end of the spectrum, 32 different processes could run on a 32-way CMT. At the other end, a single application, scaling to 32 threads—like many large commercial applications running on today's SMPs—can run on a CMT. Any point in between is also possible. For instance, you could run four application servers, each scaling to eight threads. Obviously, there are sweet spots, and it is the task of the load balancing software and operating system to appropriately leverage the hardware.

Throughput computing relies on the fact that server processors run large data sets and/or a large number of distinct jobs, resulting in memory footprints that stress all levels of the memory hierarchy. This stress results in much higher miss rates for memory operations than those typical of SPECint2000, for instance. A compounding effect is the miss cost. In about 2007, servers will already be driving over 1 Tbyte of data provided by hundreds of dual, inline memory modules. This makes the memory physically far from processors (requiring long board traces, heavy loading, backplane connectors, and so on), resulting in large cache-miss penalties.

As a result, cores are often idle, waiting for data or instructions coming from distant

Scaling the Memory Wall

Tolerating memory latency has of course been a goal for architects and researchers for a long time. Various designs have employed caches to tolerate memory latency by using the temporal and spatial locality that most programs exhibit. To improve the latency tolerance of caches, other designs have employed nonblocking caches that support load hits under a miss and multiple misses.

The use of *software prefetching* has proven effective when the compiler can statically predict which memory reference will cause a miss and disambiguate pointers.[1,2] This is effective for a small subset of applications. In particular, however, it is not very effective in commercial database applications, a prime target for servers. Notice that software prefetching can also hurt performance if it

- generates extraneous load misses by speculatively hoisting multiple loads above branches based on static analysis and
- increases instruction bandwidth.

Hardware prefetching alleviates the instruction cache pressure, using dynamic information to predict misses and discover access patterns. Its drawback (and the reason why system vendors often turn it off) is that it is hard to map a single hardware algorithm to different applications, which would require a variety of algorithms. We have observed severe cache pollution when applying a generic hardware prefetch to codes with significantly different data access patterns.

Thread-based prefetching techniques on a CMT processor use idle threads or cores to prefetch data for the main computation.[3-5]

Finally, researchers have proposed several forms of run-ahead execution; these techniques have various levels of complexity.[6-8] In general, these techniques run program instructions after long-latency instructions. They do not actually retire instructions but prefetch data into the caches. These prefetches are accurate because they are part of the instruction stream and are based on runtime information. Run-ahead execution can effectively cover L1 latency. Covering the longer latencies of remote caches and memory came at a hardware cost and some potential slowdown. New techniques such as out-of-order commit[9] are starting to appear. It remains to be seen how they will work within the context of throughput computing, but it looks promising.

Hardware scouting also runs instructions after a load miss. It does so by using mostly the same structures needed to support multithreading in a processor core. An earlier article gives a preview of how to accomplish this in a modern processor.[10] A novel aspect of scouting is that it executes instructions at a faster rate than normal execution to both ensure the timeliness of prefetches and to warm up the fetch prediction hardware and caches. Invoking and exiting out of scouting is a zero-cycle operation to avoid slowdowns from unsuccessful scouting events. This is an important implementation aspect that dictates the organization of important pipeline stages (this has been our experience). We conclude that even though architects can no longer mostly rely on faster transistors and faster wires to provide an easy performance scaling, certain ways of using the transistor budget compensate for the lack of scaling; these reach high levels of performance, especially in the context of throughput computing.

References

1. C.-K. Luk and T.C. Mowry, "Compiler-Based Prefetching for Recursive Data Structures," *Proc. 7th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 96), ACM Press, 1996, pp. 222-233.
2. T.C. Mowry, M.S. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proc. 5th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 92), ACM Press, 1992, pp. 62-73.
3. S. Chaudhry and M. Tremblay, "Method and Apparatus for Using an Assist Processor to Pre-fetch Data Values for a Primary Processor," US Patent 6,415,356.
4. J.D. Collins et al., "Dynamic Speculative Precomputation," *Proc. 34th Ann. ACM/IEEE Int'l Symp. Microarchitecture* (Micro-34), IEEE CS Press, 2001, pp. 306-317.
5. C. Zilles and G. Sohi, "Execution-Based Prediction Using Speculative Slices," *Proc. 28th Ann. Int'l Symp. Computer Architecture* (ISCA 01), IEEE CS Press, 2001, pp. 2-13.
6. J. Dundas and T. Mudge, "Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss," *Proc. 1997 Int'l Conf. Supercomputing* (SC 97), IEEE CS Press, 1997, pp. 68-78.
7. R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi, "Dynamically Allocating Processor Resources Between Nearby and Distant ILP," *Proc. 28th Ann. Int'l Symp. Computer Architecture* (ISCA 01), IEEE CS Press, 2001, pp. 26-37.
8. O. Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," *Proc. 9th Int'l Symp. High Performance Computer Architecture* (HPCA 03), IEEE CS Press, 2003, pp. 129-140.
9. A. Cristal, D. Ortega, J. Llosa, and Mateo Valero, "Out-of-order Commit Processors," *Proc. 10th Int'l Symp. High Performance Computer Architecture* (HPCA 04), IEEE CS Press, 2004, pp. 48-59.
10. J. Niccolai, "Sun Adds Rock to its UltraSPARC Road Map," *Computerworld*, 12 Feb. 2004; http://www.computerworld.com/hardwaretopics/hardware/story/0,10801,90145,00.html.

memory locations. Depending on the processor's latency tolerance, "distant memory" can be the first few levels of caches, external caches, other processors' caches, and/or main memory. As we will describe later, even very aggressive out-of-order processors will be idle, waiting for memory data, for hundreds of cycles.

CMT allows the processor to switch on-chip threads into the pipeline after it discovers events causing long stalls; Figure 1 shows this strategy. CMT uses the fact that some of the outer structures (those further from the core) are often architected for full pipelining and yet are idle most of the time. Examples of such structures include memory controllers, I/O controllers, and networking accelerators. CMT processors share these structures among cores and threads, resulting in much better utilization. Alternatively, we find that adding full pipelining to non-pipelined structures, such as memory controllers, cyclic redundancy checkers, or large MMUs adds slightly to the area and yet allows full sharing at very little performance cost. Figure 2 shows how cores and threads overlap spatially and temporally to achieve a large amount of work (high throughput).

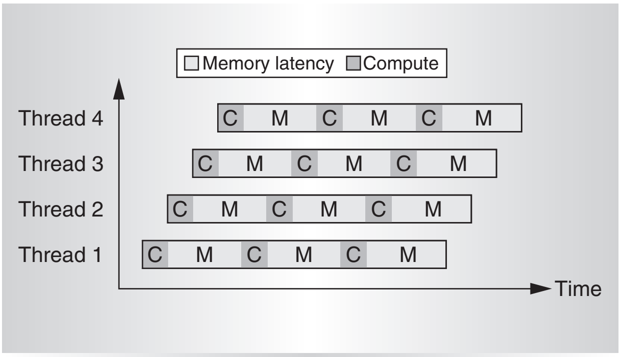

Figure 1. Systems with multithreaded cores hide stall cycles by switching alternate threads and processes into the pipeline.

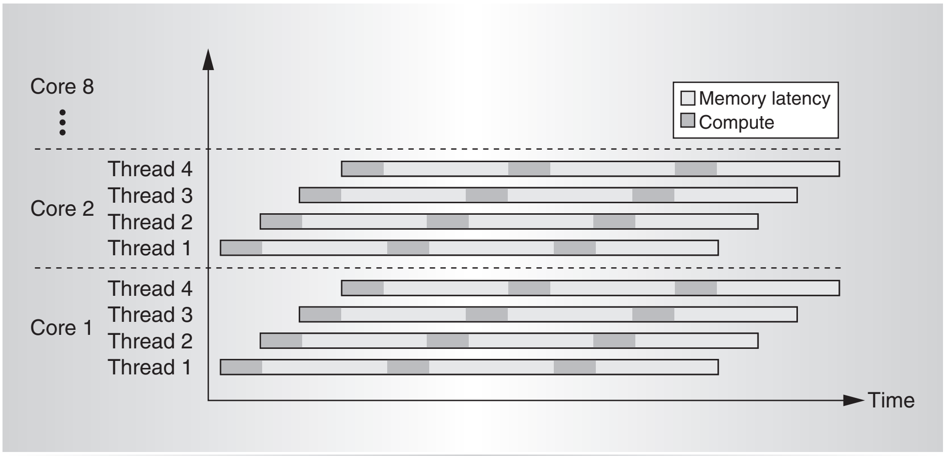

Figure 2. A 32-way CMT combines multithreaded cores into an on-chip SMP to deliver high throughput.

Throughput computing is as much about the consolidation of on-chip resources as it is about consolidating assets at the server granularity. This strategy accommodates data sharing among processors in a large SMP on-chip or across a few CMT chips with much shorter latencies. For horizontally scaled applications, this permits consolidation of multiple copies of the operating system and the applications into one copy of the operating system and a single physical copy of the application (with multiple virtual copies). This consolidation can result in large savings in terms of memory, often the dominant cost in these platforms.

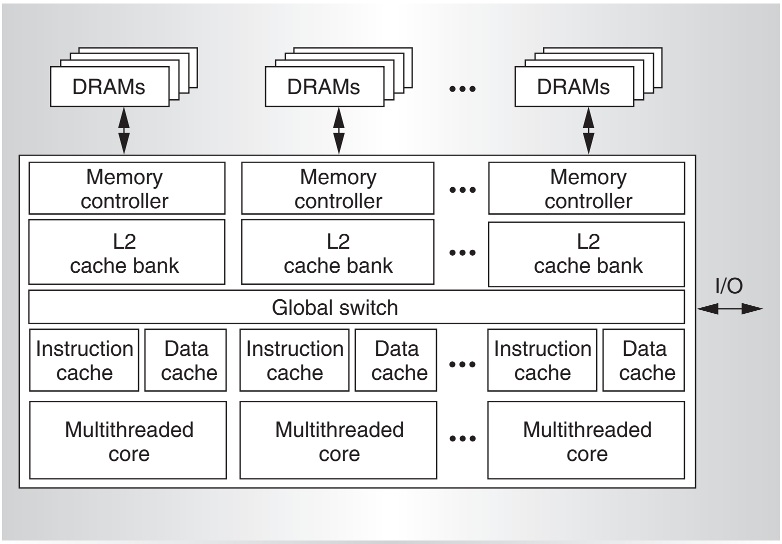

Figure 3. Block diagram of a typical CMT.

Typical CMT

In Figure 3, we show a block diagram for a

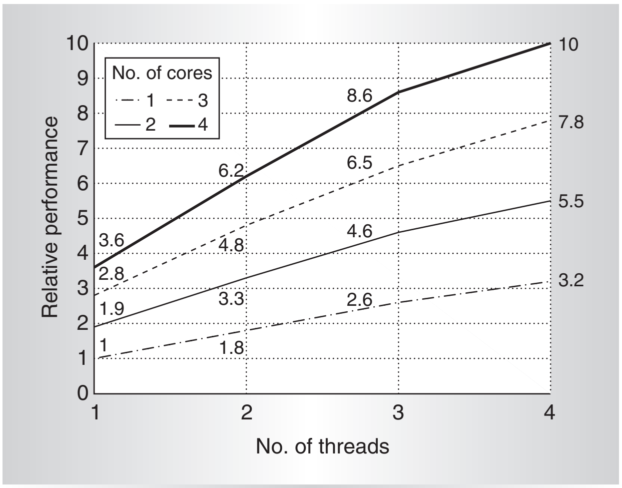

Figure 4. Impact of cores and threads on lightweight cores.

typical CMT processor, one that we have been using for simulation and analysis over the years. We will use it throughout this article. It is fairly generic and yet maps nicely onto the Niagara processor, for instance.[6] In this generic processor, a few threads (four for Niagara) share a pipeline and form a core. For a T-threaded core, we assume T copies of the register file and T copies of the internal processor registers. We assume that other inner structures, such as the level-one (L1) instruction cache, data cache, and translation lookaside buffers, are shared within a core but not across cores.

This model views the memory subsystem as a single logical structure. In our diagram, a global switch connects cores and the shared logical memory. The on-chip L2 cache has banks for increased bandwidth. Figure 3 shows the same number of banks as the number of cores, but that need not be the case. Each bank of the L2 cache has attached memory controllers. We use this model to study the impact of threading and multiple cores on overall performance.

Multithreading and multicore

In trying to architect a core for a CMT chip, there is an inherent tension between the complexity of each core and number of cores that can fit on a die. If you choose to implement a lightweight core or a core that does not target high instruction level parallelism (ILP) and high clock rate, single-thread performance will suffer, but high overall throughput is achievable through replicating many of them.

The Niagara core is a good example of a power-efficient multithreaded core.[6] It uses a simple one-issue, in-order pipeline with little speculation, which translates into a small area, low power, and a fairly easy core to multithread.

A complex core will require more effort to implement, likely result in a more expensive part, and likely draw more power. On the other hand, it can offer substantial performance gain.

Both simultaneous multithreading (SMT) and vertical threading (VT) enable a core to support multiple threads. Vertical threading is simpler because it has the restriction that, on a given cycle, the processor can only issue instructions from a single thread. Obviously, VT is less able to optimize pipeline resource utilization.

Lightweight cores become more attractive when coupled with multithreading. A core using SMT or VT can take advantage of otherwise idle resources while not significantly impacting other threads. Since lightweight cores have a relatively low resource utilization per thread, such cores can support many threads before per-thread performance starts to degrade. Supporting four threads can scale total performance by as much as 3.2× on a large database benchmark, as Figure 4 shows. The figure also shows that a lightweight core with four threads has greater performance than three single-threaded cores, a very good trade-off.

These large performance improvements might appear surprisingly high when compared with other commercial processors. The key to achieving them is to architect cores from the very beginning to support multithreading (as opposed to adding multithreading to an existing core not explicitly built for multithreading). As an example, in the context of throughput computing, there is little to gain from trying to access the L1 cache in two cycles (versus three). This relaxation permits the removal of complex and power hungry hardware, which has the benefit of reducing the area and power of a core. However, it also lowers the initial single-thread performance, anticipating that the multithreaded performance will make up for this deficit.

A heavyweight core will improve single-thread performance, but generally reduces the number of cores that will fit onto a chip. Also, compared to a lightweight core, a heavier core will typically have higher power consumption. Nevertheless, the judicious addition of complexity can increase performance per square millimeter as well as performance per watt. Experiments show that an otherwise fairly lightweight core with hardware scouting capabilities (which we discuss later), can outperform a truly lightweight core by 4 to 10×.

Multithreading is also a profitable addition to a heavyweight core. Although a heavier core is better able to optimize per-thread pipeline utilization and is better at extracting ILP than a lighter core, there still remains significant opportunity for an additional thread to utilize otherwise idle resources.

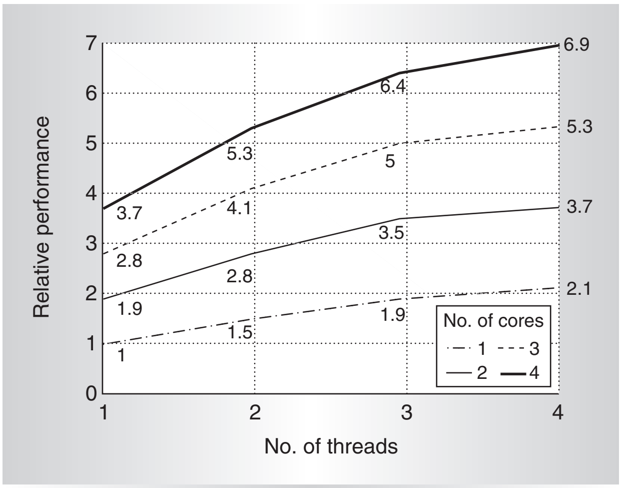

Figure 5. Impact of cores and threads on heavyweight cores.

We have observed that a two-thread core achieves a 1.4 to 1.5× improvement in throughput. This boost in performance costs only a relatively small increase in area (about 10 percent). The additional improvement from a four-thread core starts to diminish, yielding another 1.3 to 1.4× improvement. This incremental performance comes at an incremental cost of about 15 percent for the state of the additional two threads and the slightly bigger state machines.

Multicore technology can help to extract thread-level parallelism in either lightweight or heavyweight core designs. By putting four lightweight cores on a die, we observed up to a 3.6× increase in performance (see Figure 4). Similarly, as Figure 5 shows, four heavyweight cores on a die showed improvements of up to 3.7×. Therefore, almost regardless of the type of core being used, throughput processors have much to gain from replicating cores and organizing them efficiently.

Furthermore, the gains achieved by replicating cores are largely independent of the gains obtained by multithreading. We observed substantial performance improvement by putting several multithreaded cores on a die. For example, a chip using four lightweight cores, each having four threads per core, delivered a combined 10× performance. A heavier multithreaded core was also able to yield a 6.9× improvement. Of course, this scaling is based on the already impressive 4 to 10× advantage of the heavier core.

We generated the data in Figures 4 and 5 to understand scaling, sharing, inflexion points, and so on. To isolate these effects, we assume that

- all threads on a core share the L1 caches,
- all threads on a chip share the TLBs,
- the number of threads or cores does not affect the access times to caches, and
- that the processor connects directly to memory through multiple memory controllers (that is, there is no L3 cache).

Single-thread performance

CMT processors built from heavyweight cores can provide good single-thread performance. One choice to consider is a traditional out-of-order core. Such processors have been very popular and fairly successful in running commercial applications because they can execute past load misses and overlap multiple misses. This reduces the effective memory latency, resulting in a high number of instructions per cycle (IPC).

The ability to overlap misses is limited by the window size and the amount of speculation supported. A small window limits the

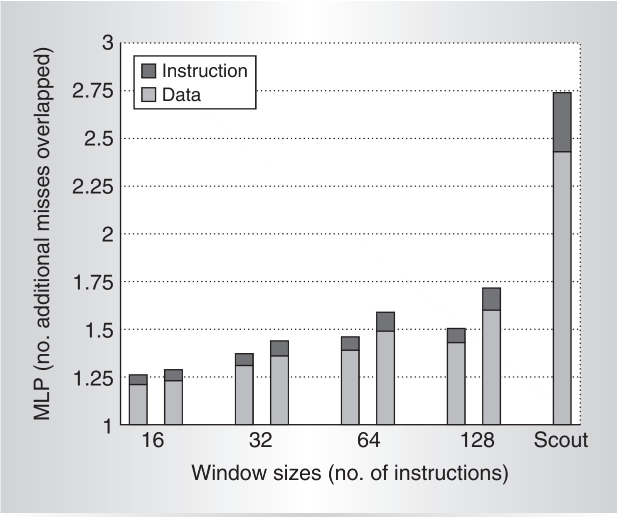

Figure 6. Database memory parallelism. The left bar of each set is for a moderately speculative, out-of-order core; the right bar is for a core with an aggressive issue policy.

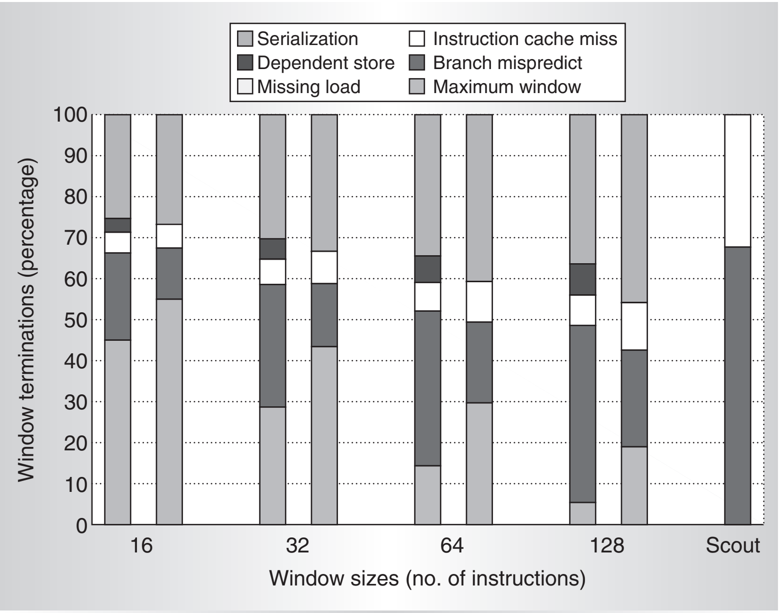

Figure 7. Window termination conditions. The left bar of each set is for a moderately speculative, out-of-order core; the right bar is for a core with an aggressive issue policy.

number of instructions that are potential candidates for overlap. For example, a 32-instruction window limits the search for the next load miss to 31 instructions following the first load miss. Unfortunately, as we discussed previously, the growing gap in the cycle time of dual inline memory modules and processors has resulted in memory accesses that can take hundreds of processor cycles to complete. Thus, to "tolerate" this long latency and overlap multiple load misses, an out-of-order processor needs a very large window and highly speculative execution. This increases the core area substantially; some structures grow quadratically with window size. Growing this window size to 64, 128, or even 256 instructions through traditional methods runs counter to the desire to build a core small enough to fit many of them on a single chip. The data on the UltraSPARC V core, a 128-instruction-deep out-of-order processor, showed that it was about nine times as big as the four-issue in-order core in UltraSPARC I. The power associated with the large associative structures (content-addressable memories) and broadcast networks also make traditional out-of-order cores unattractive for throughput computing.

The amount of speculation that fits on a chip limits the ability to overlap multiple load misses. For example, if a store address is unknown (because it depends on a load miss), and the processor does not support memory disambiguation for possible read-after-write hazards of subsequent loads, then these loads cannot be candidates for miss overlap. Also, atomic operations typically stall the decode stage of the pipeline until all prior instructions have completed. This too prevents subsequent loads from becoming candidates for miss overlap.

Lastly, some programs do have independent instructions but these lie farther along in the instruction stream than a limited hardware structure can handle. In such programs, a processor with even a reasonably large window size of as much as 128 instructions will not find other independent loads that it can overlap to memory.

Moreover, aggressive speculation only adds to the complexity and area. The design must control the aggressiveness using predictors and rewind mechanisms, and it must implement structures to correct any miss-speculation.

The following figures illustrate the degree of MLP obtained when running a database workload with different core architectures. In Figure 6, the *y*-axis shows the number of additional misses overlapped to memory from a 2-Mbyte unified L2 cache. On the *x*-axis, we show different window sizes and different degrees of speculation. The first bar in each pair represents a moderately speculative, out-of-order core; the second represents an aggressive issue policy. In particular, the aggressive policy allows stores to issue out of order with respect to other stores and allows branches to issue out of order with respect to other branches.

Figure 7 identifies the cause that stopped the processor from examining subsequent instructions for issue and thus prevented it from discovering more independent load misses for overlapping. Serialization caused by atomic instructions begins to dominate as the cause.

Hardware scout

To achieve high MLP and thus high single-thread performance, we must issue independent memory requests that can be found fairly far downstream from an initial load miss. Furthermore, our technique for doing this must avoid the area, complexity, and window size restrictions inherent in the traditional out-of-order approach.

A hardware scout processor achieves MLP by creating a resumable hardware checkpoint of program state on encountering a load miss. The core can then continue issuing instructions arbitrarily far along in the instruction stream until the memory subsystem returns the load data. When the data returns, the processor resumes execution from the checkpointed state. Naturally, memory operations that the processor reissues on the second pass have the advantage of having been previously prefetched into the primary caches.

The principal advantage to this approach is that the scouting distance is proportional to the number of cycles elapsed waiting for the load miss data. The size of on-chip structures—such as the size of an out-of-order issue window—does not limit scouting. Moreover, hardware scouting relies only on redeploying the existing pipeline resources, a more area efficient approach than having an autonomous hardware prefetch engine.

In the example in Figure 8, we see an initial load to register 7 missing and becoming the cause of a checkpoint. The entire sequence of instructions in the gray area then issues speculatively. This approach uses a separate register file, called the shadow register file, to make speculative computational results available for further address computations while the main register file safely retains the checkpoint state.

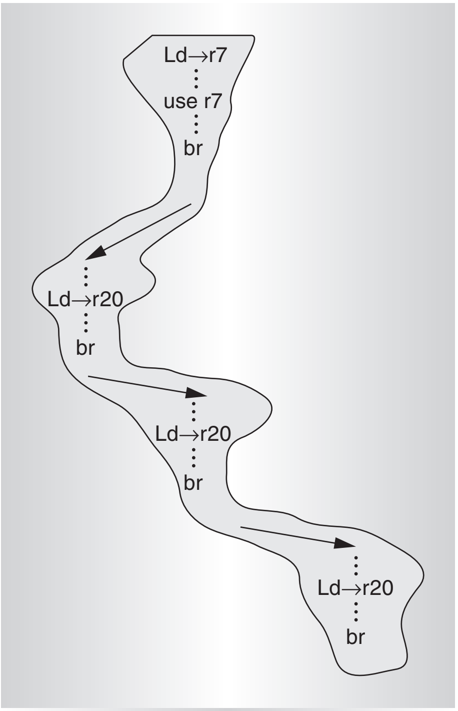

Figure 8. Example of how a hardware-scouting processor handles a load miss.

Of course, a subsequent load might have an address that depends on the original load that missed. Or, for that matter, it might depend on a use of the original load miss, another load miss, a use of a use of a load that missed, or so on. Since bandwidth is precious, we do not want to issue memory operations having nonsensical addresses to the caches. To accomplish this, each architectural register has an associated *not there* (NT) bit. When a load misses, the NT bit is set for the destination register. Additionally, the NT bit is set for the destination register of any instruction that has a source register NT bit set. That is, the destination's NT bit is assigned a logical OR of the source registers' NT bits.

It is worth noting that the NT bits do not propagate indefinitely. Once a register is redefined and the processor reuses it in the program, it clears the register's NT bit. This happens automatically because of the OR operation previously described. An instruction whose source registers are available (and is not itself a missing load) will clear the NT bit for the destination register regardless of

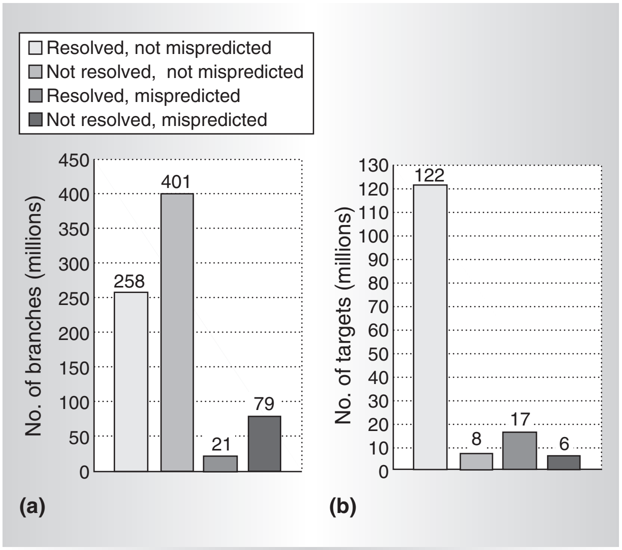

Figure 9. Simulation results for the database workload, showing four classes of branch (a) and jump target (b) prediction.

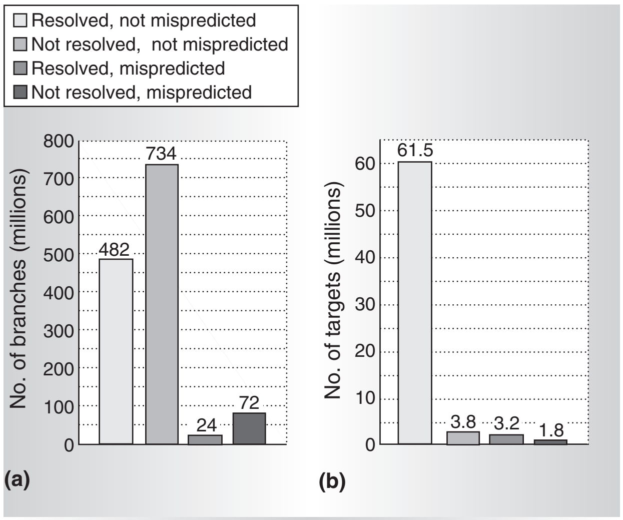

Figure 10. Simulation results for SPECint2000, showing four classes of branch (a) and jump target (b) prediction.

whether it had been previously set.

Besides load misses, hardware scouting ameliorates other conditions that are traditionally stalling in nature. For example, a full store buffer is a stall condition for both traditional in-order and out-of-order designs. This becomes, for us, just another reason to take a checkpoint and begin hardware scouting. Since the processor uses store addresses to prefetch cache lines, the scouting will help ensure that the store buffer can drain more quickly for stores that will be reached architecturally once execution resumes from the checkpoint.

Note that the computational results calculated while scouting never commit to architectural state. This permits the freedom to perform optimizations beneficial to the common cases without concern for the correctness of a rare situation. As an example, data from stores that fit in the store buffer bypass correctly to loads, while stores that overflowed the store buffer capacity do not. In effect, we are speculating that loads leading to address computation do not have read-after-write hazards with the overflowed stores.

The processor will encounter branches and other control transfer instructions while scouting. Many of these control dependencies are resolvable, that is, independent of outstanding load misses. Thus, regardless of whether the fetch unit predicted them correctly, the scout thread can proceed down the correct path. The mispredicted control transfer instructions result only in a redirection of the fetch unit, as is normally done.

Upon encountering a control dependency that is not resolvable (the NT bit is set on the condition code register), hardware scouting continues executing along the fetch-unit-predicted path. This usually results in following the correct path. Note that there are no correctness issues associated with executing down the wrong path since instructions only write to the shadow register file, and the processor will eventually restart execution from the checkpoint. In fact, our results show that the more difficult to predict branches are usually associated with small if-else programming constructs that quickly return scout execution to the correct path. The experimental simulation results described in this article do not include this effect because our trace-driven methodology does not contain the wrong path instructions. Hence, we simply do not model scouting through these control dependencies.

Atomic operations (lock acquisition) and other memory barriers are a significant performance inhibitor to traditional methods for extracting ILP. In contrast, a scout-capable processor can scout past these barriers and prefetch data close to the requesting processor. Because software locks are typically uncontested, it is valuable to request the lockprotected cache lines. This is especially true because it was likely a remote processor that last used the shared data. Thus, the loads shortly following the barrier are often fertile territory for discovering MLP.

Because control speculation is our primary possibility for miss-speculation, we have some simulation results showing how well the scouting hardware handles control dependencies. For these results, we collected an instruction trace of 1.2 billion instructions from a commercial database workload and 500 instruction trace samples containing 1 million instructions each for both SPECint2000 and SPECfp2000. Using simulations, we classified each branch or jump instruction into one of four categories: resolved, correctly predicted; unresolved, correctly predicted; resolved, mispredicted; and unresolved, mispredicted.

Branch prediction is very good overall, 87 to 98 percent correct. Furthermore, between 37 to 72 percent of branches are resolvable during speculative execution. Branch prediction for the unresolvable branches is 84 to 97 percent accurate. Overall then, the hardware can scout in the correct direction 90 percent of the time for the database workload, as Figure 9a shows; for the SPECint2000 and SPECfp2000 codes the prediction is even better, as Figures 10a and 11a show.

The jump target predictions illustrated in Figures 9b, 10b, and 10c show that scouting is only 4 percent incorrect in the worst case. Of course, this is no accident. When architecting a hardware scout processor from a clean slate, you must consider the importance of accurate predictions, then size buffers accordingly. Moreover, you must pay attention to how predictions made while scouting interact with predictions made while running the main thread. As an example, it is advisable to make the return address stack part of the checkpoint to properly predict method returns after having scouted.

The histograms in Figure 12 show how deep into the instruction stream scouting can proceed. The height of each bar indicates how many scouting events resulted in executing the number of instructions shown on the *x*-axis. The scale on the *x*-axis is from 0 to 1,000 instructions. These histograms reveal three natural groupings in the modeled configura-

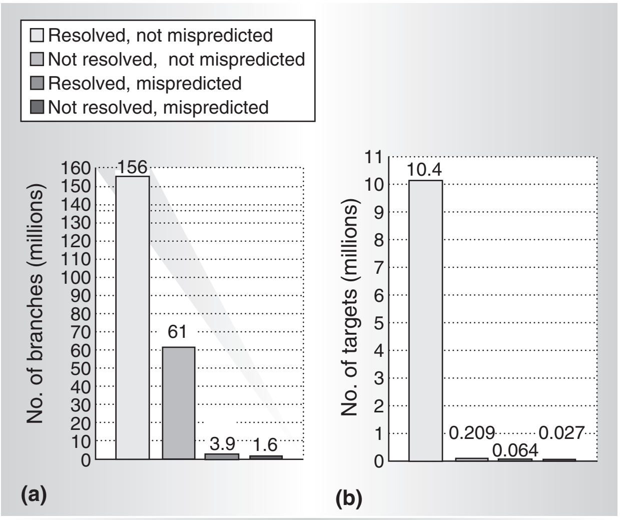

Figure 11. Simulation results for SPECfp2000, showing four classes of branch (a) and jump target (b) prediction.

tion; they correspond to the latencies of the L2 cache, the L3 cache, and memory. Clearly, the L2 cache satisfies the majority of cache misses, so we created the second graph to better show the clustering at the higher latencies.

For the database workload, we see that the scouting depth is more smoothly distributed. Because this multithreaded program has more primary cache misses, queuing delays in the memory subsystem somewhat blur the latency clusterings. The second histogram for the database workload shows the *y*-axis scaled logarithmically. The last bar is artificially high because it includes all distances greater than 1,000 instructions.

Scouting can progress 500, 600, even 700 instructions or more. Naturally, this relates to the memory latency—the longer it takes to satisfy a load request, the farther execution can proceed. What is not immediately obvious is the fact that dependent instructions (instructions having one or more source registers NT) can issue faster than independent ones. Recall that the only operation required of a dependent instruction is the single-bit ORing of the source NT bits to set the destination NT bit. Thus, these instructions can group without regard to pipeline resources such as arithmetic logic units, memory pipes, and so on.

Additionally, the issue of dependent instructions can ignore all data dependencies. For example, if one source register is NT, there is no point in stalling to wait until a multicy-

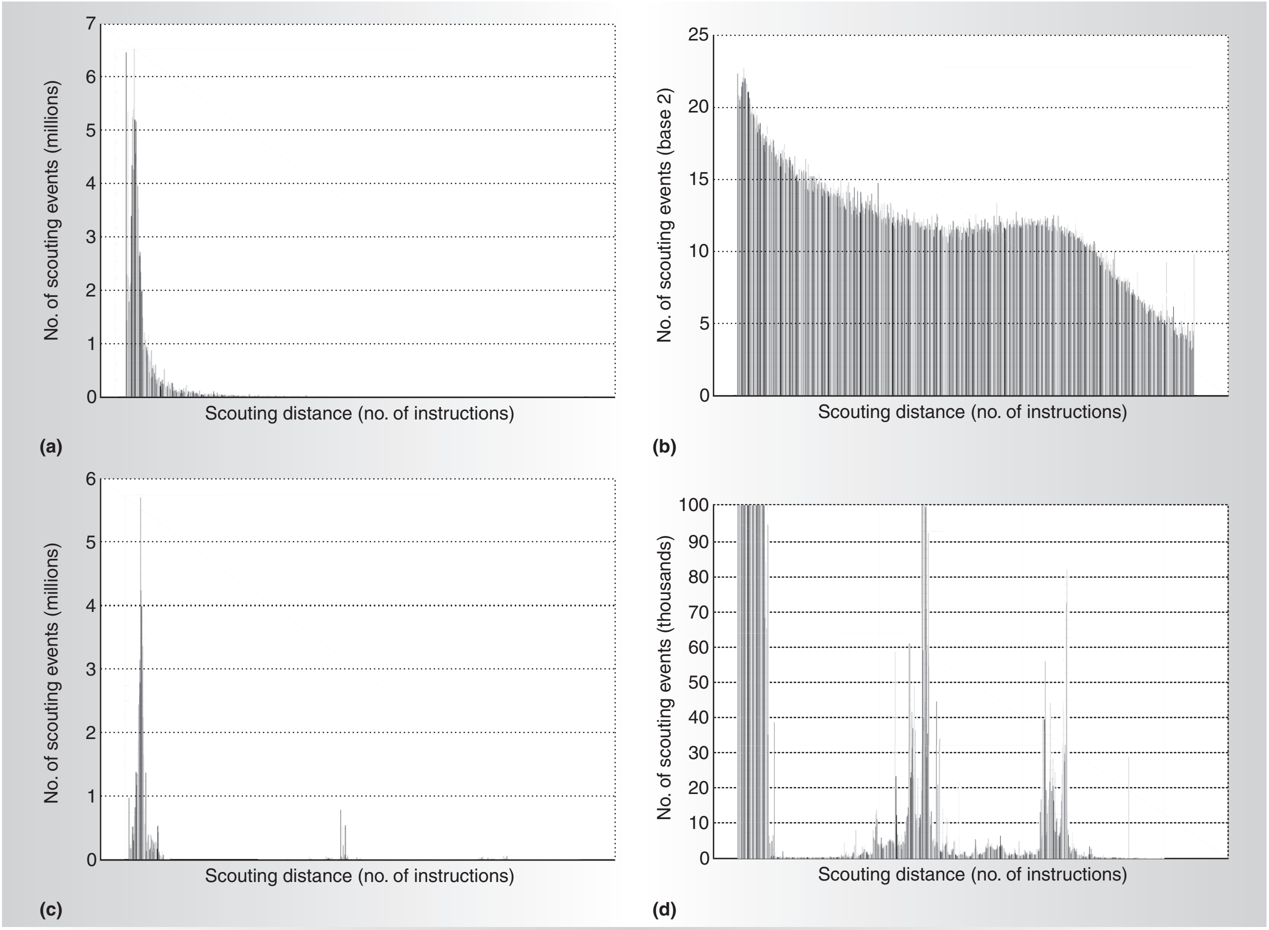

Figure 12. Histograms showing the incidence of scouting on the database workload, for all latencies plotted linearly (a) and plotted semi-logarithmically (b). Similar SPECint2000 histograms are for all latencies (c) and for just the higher latencies (d). These show the natural three latency groupings.

cle instruction produces the other source register. In fact, if one of a producer's sources is NT, even producers and consumers can issue together in the same cycle by forwarding the NT result of the producer to the consumer.

The value of relaxing the issue rules for dependent instructions is that the processor encounters load misses more quickly. This helps ensure that the memory system will have responded with the data in a timely fashion. If it requires *N* cycles to reach a missing load instruction during normal execution, the scouting hardware will reach that same instruction in fewer than *N* cycles. This allows more time for the data to return to the pipeline.

The key to high performance in modern server microprocessors is the ability to overlap long cache misses. To quantify this effect, consider the metric of MLP, which we define as the average number of outstanding memory requests over the time period when there is at least one request outstanding.[12] Figure 13 show the overall MLP for our benchmarks as well as the MLP when we consider only loads, stores, or prefetch instructions in isolation. These figures show the relative improvement in MLP as compared to a moderately aggressive, out-of-order processor.

Clearly, scouting's effect on MLP is dramatic. Overall, it increases MLP for the database workload by more than 20 percent; for SPECint2000, 32 percent; and for SPECfp2000, 70 percent. Considering loads alone, scouting increase MLP by more than 49 percent. Loads are arguably the most critical to overall performance, but stores also matter because they do accumulate in a finitesize store buffer. Again, we see that scouting increases the MLP of stores by approximately 20 percent. The figures also show prefetch MLP, but the result for the database workload is anomalous in that scouting decreases MLP. Our analysis shows that scouting finds the prefetches early and also ends them early. Furthermore, the number of prefetch instructions in the database trace is low, so the effect is actually small in absolute terms.

The cycles spent in scouting benefit the architectural thread in several ways. First, as mentioned, load misses are issued to the memory subsystem and so you can think of scouting as a sophisticated hardware prefetch engine that not only prefetches exactly what the processor will need, but also does so through the reuse of existing execution resources. The processor can use this "prefetch engine" precisely when it is most useful—when the pipelines would otherwise stall. Moreover, the address computation uses the same instructions and register values that would otherwise later result in a load miss.

Secondly, hardware scouting primes other on-chip structures. It warms up the instruction cache, for example, with instructions that the processor will need in the immediate future. Thus, future instruction cache misses overlap with the current data cache miss. Additionally, the branch prediction array and jump target buffer both learn how to correctly predict upcoming control transfers.

For the case of a classic two-bit branch predictor, a designer ought to account for whether the branch is resolved speculatively or architecturally when updating the counter. In particular, the strongly taken backward branch at the end of a loop should not be updated to a not-taken state upon architecturally exiting the loop. (A naïve implementation might update this end-of-loop branch twice, once during scouting and again when reaching it architecturally. Such a double update might first update strongly taken to weakly taken and then to not-taken, an undesirable outcome.)

Scouting also results in higher performance. Figure 14 illustrate the improvement in IPC for various cache sizes. For example, the database workload shows a 40 percent improvement in performance for a 512-Kbyte L2 cache. Or, comparing points having approximately equal performance, a 1-Mbyte cache offers the performance of an 8-Mbyte one. Or, a 4-Mbyte cache offers the performance of a 16-Mbyte cache.

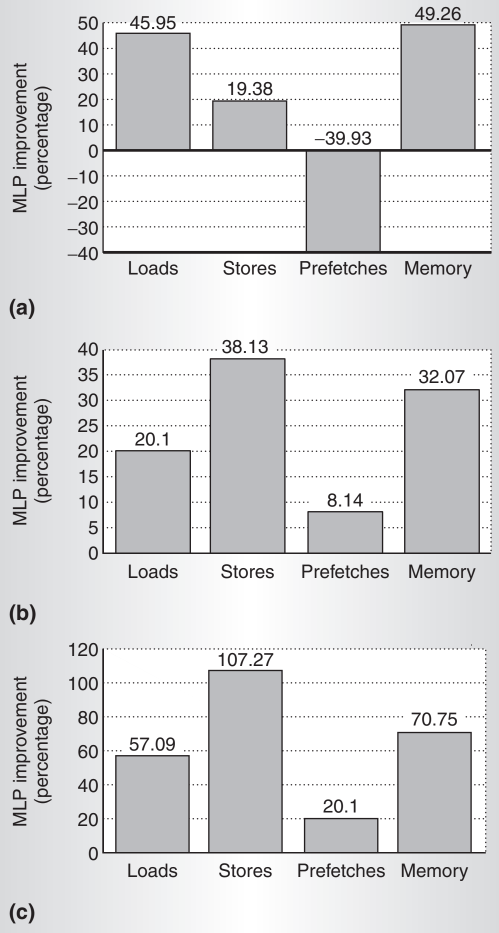

Figure 13. Percentage MLP improvement from scouting for the database (a), SPECint2000 (b), and SPECfp2000 (c) workloads.

For SPECint2000, scouting yields a 12 percent performance boost for a 512-Kbyte cache, an attractive gain considering the low miss rate of these benchmarks. Similarly, a 1-Mbyte cache in a scouting processor is as effective as a 2-Mbyte cache in a processor incapable of scouting. The SPECfp2000 workload shows some of the most dramatic gains. With a 512-Kbyte cache, scouting increases SPECfp2000 performance by 34 percent. With a 1-Mbyte cache, the scouting processor's performance exceeds that of a processor without scouting that has a 64-Mbyte L2 cache.

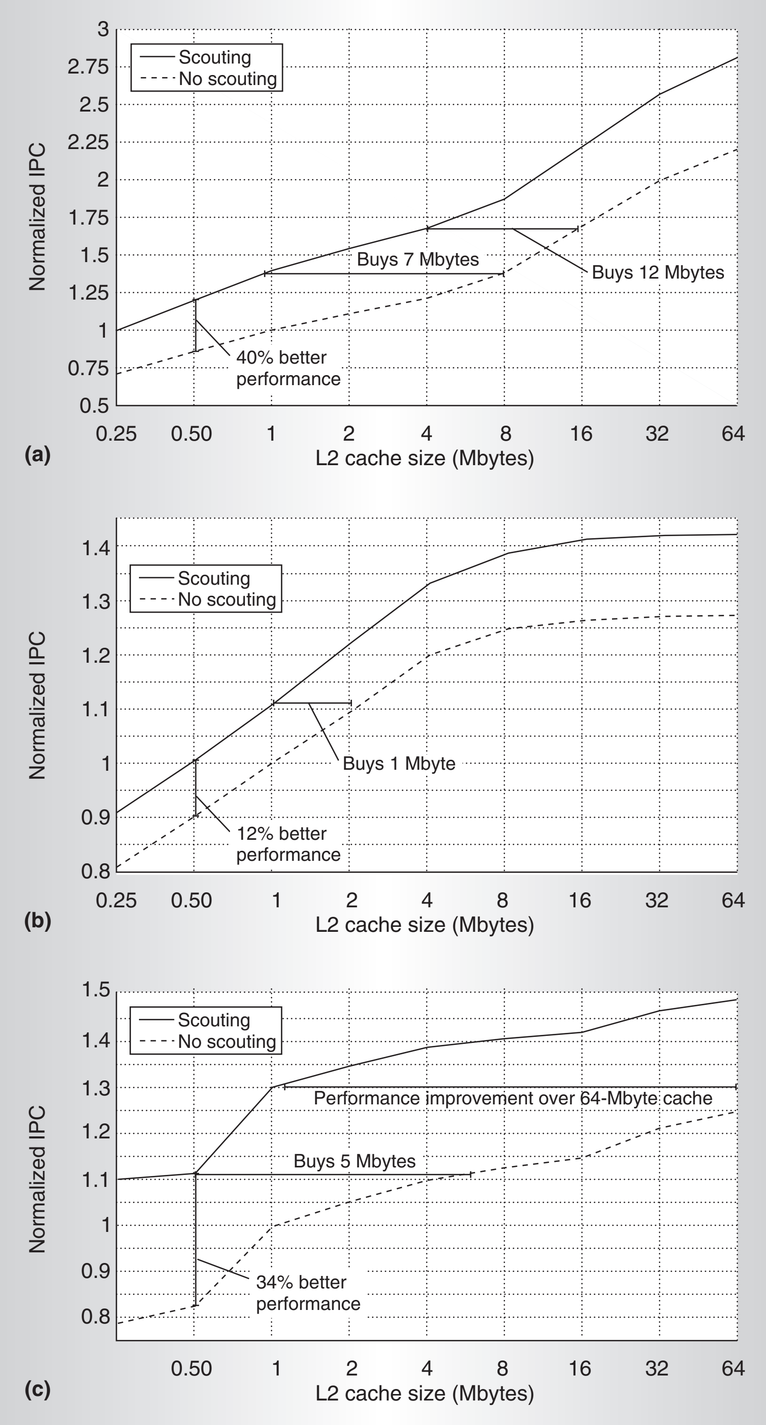

Figure 14. Normalized IPC improvement from scouting for the database (a), SPECint2000 (b), and SPECfp2000 (c) workloads. IPC improvement varies with cache size.

CMT processors offer a way to significantly improve the performance of computer systems. The return on investment for multithreading is among the highest in computer microarchitectural techniques. If you design a core from scratch to support multithreading, gains as high as 3× are possible for just a 20 percent increase in area. Few microarchitectural techniques can offer this level of improvement for the same additional area. Likewise, the availability of hundreds of millions to a few billion transistors permits building a network of multithreaded cores that can also offer excellent scalability. Here again, the key is to design the network with CMT in mind. In other words, the many wires and high frequency available on chip—as opposed to on a board or backplane—should dictate what the network will look like. If architected properly, linear scalability for a large set of applications is possible.

Even with throughput performance as the main target, we have shown that the microarchitecture necessary to support threads on a CMT can also achieve high single-thread performance. Hardware scouting, which Sun is implementing on the Rock microprocessor, can increase the single-thread performance of applications by up to 40 percent. Alternatively, scouting is a technique that makes the on-chip caches appear much larger, performance wise. Finally, it is a performance robustness technique, making up for code tailored for different on-chip cache sizes or even a different number and levels of caches.

We have attempted to show that to fully exploit the promises of multithreading and multicores, you must architect for it from the very beginning. In the future, we will show how going down the path of CMT and hardware scouting leads to a new execution model that does not fit into the normal classification of in-order or out-of-order processors. We will show how to attain even higher single-thread performance by building on top of hardware scouting and combining some of our earlier work on speculative multithreading.[2] MICRO

References

1. M. Tremblay, "MAJC: An Architecture for the New Millennium," Hot Chips 11, 1999; http://www.hotchips.org/archives/hc11/3_Tue/hc99.s8.2.Tremblay.pdf.
2. M. Tremblay et al., "The MAJC Architecture: A Synthesis of Parallelism and Scalability," *IEEE Micro*, vol. 20, no. 6, Nov.-Dec. 2000, pp. 12-25.
3. S. Kapil, "Gemini: A Power-Efficient Chip Multi-Threaded UltraSPARC Processor," Hot Chips 15, 2003; http://www.hotchips.org/archive/hc15/pdf/12.sun.pdf.
4. Q. Jacobson, "UltraSPARC IV Processors," *Microprocessor Forum*, In-Stat, 2003.
5. D. Greenley, "Sun UltraSPARC IV+ Processor," *Microprocessor Forum*, In-Stat, 2004.
6. P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, vol. 25, no. 2, Mar.-Apr. 2005, pp. 21-29.
8. J. Tendler et al., "Power4 System Microarchitecture," *IBM J. Research and Development*, vol. 46, no. 1, Jan. 2002, pp. 5-25.
9. J. Clabes et al., "Design and Implementation of the Power5 Microprocessor," *Proc. 41st Ann. Conf. Design Automation* (DAC 04), ACM Press, 2004, pp. 670-672.
10. S. Kunkel et al., "A Performance Methodology for Commercial Servers," *IBM J. Research and Development*, vol. 44, no. 6, Nov. 2000, pp. 851-872.
11. K. Flautner et al., "Thread Level Parallelism and Interactive Performance of Desktop Applications," *ACM SIGPLAN Notices*, vol. 35, no. 11, Nov. 2003, pp. 129-138.
12. Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture Optimizations for Exploiting Memory-Level Parallelism," *Proc. 31st Ann. Int'l Symp. Computer Architecture* (ISCA 04), IEEE Press, 2004, pp. 76-89.

Shailender Chaudhry is a distinguished engineer at Sun Microsystems where he is chief architect of the Rock processor. His experience includes work on the MAJC architecture, picoJava II, microJava, and space-time computing. His research interests include computer systems architecture, algorithms, and protocols. Chaudhry has a BS and an MS in computer engineering from Syracuse University.

Paul Caprioli is a senior staff engineer at Sun Microsystems where he is the Rock core architect and leads the performance modeling group. His research interests include computer architectures and high-performance technical computing. Caprioli has an MS and a PhD in mathematics from Rensselaer Polytechnic Institute.

Sherman Yip is an engineer at Sun Microsystems where he is a member of the Rock architecture team and responsible for performance modeling. His research interests include processor simulation, visualization, and Java performance and design. Yip has a BS in computer science from the University of California at Davis.

Marc Tremblay is a Sun Fellow and distinguished engineer at Sun Microsystems, where he is chief architect of the Scalable Systems Group. His research interests include chip multiprocessing, chip multithreading, speculative multithreading, and assist threading. Tremblay has an MS and a PhD in computer science from UCLA and a BS in physics engineering from Laval University in Canada. He holds 97 US patents in various areas of computer architecture. He is a member of the IEEE and the ACM.

Direct questions and comments about this article to Marc Tremblay, Sun Microsystems Inc., 430 North Mary Ave., Sunnyvale, CA 94085; marc.tremblay@sun.com.