Dual-Thread Speculation:
Two Threads in the Machine are Worth Eight in the Bush

Fredrik Warg and Per Stenstrom
Chalmers University of Technology

Dual-Thread Speculation: Two Threads in the Machine are Worth Eight in the Bush*

Fredrik Warg and Per Stenstrom
Department of Computer Science and Engineering
Chalmers University of Technology, SE-412 96 Goteborg, Sweden
{*warg,pers*}@*ce.chalmers.se*

Abstract

As chip multiprocessors with simultaneous multithreaded cores are becoming commonplace, there is a need for simple approaches to exploit thread-level parallelism. In this paper, we consider thread-level speculation as a means to reap thread-level parallelism out of application binaries. We first investigate the tradeoffs between scheduling speculative threads on the same core and on different cores. While threads contend for the same resources using the former approach, the latter approach is plagued by the overhead for inter-core communication. Despite the impact of resource contention, our detailed simulations show that the first approach provides the best performance due to lower inter-thread communication cost.

The key contribution of the paper is the proposed design and evaluation of the dual-thread speculation system. This design point has very low complexity and reaps most of the gains of a system supporting eight threads.

1 Introduction

The difficulties involved in exploiting instruction-level parallelism (ILP) and increasing the clock frequency has led to a paradigm shift in processor architecture from complex single-processor chips to chip multiprocessors (CMPs) containing multiple moderately complex processor cores. Additionally, in IBMs Power5 [5] and SUNs Niagara [6], each core can run several threads concurrently using simultaneous multithreading (SMT) [16]. We refer to such systems as *multithreaded chip architectures* (MCA). MCAs can obviously unleash much thread-level parallelism, but it is presently unclear how existing single-threaded applications can benefit from MCAs.

Several researchers have considered thread-level speculation (TLS) as a means to transparently uncover thread-level parallelism from chip multiprocessors [2, 4, 7, 10, 14, 15, 12] and simultaneous multithreaded processors [1, 11, 9]. TLS does this by executing e.g. consecutive loop iterations speculatively in parallel assuming there are no loop-carried dependences. If the speculation succeeds, the speculative state is committed to memory in program order when the loop terminates. If it fails, the misspeculated threads are squashed and re-executed, serially. This paper focuses on two important issues regarding how to best support thread-level speculation on an MCA. The first question is where speculative threads should be scheduled: On the same SMT core, or on other cores? Since threads share the same core, and typically the first level cache, thread starts and restarts can be more efficiently implemented on an SMT processor. On the other hand, these threads contend for the same execution units and cache, which might impact the performance negatively. Conversely, scheduling the threads on different cores will potentially reduce resource contention but may suffer from the higher inter-core communication overheads. Our first contribution is our finding that scheduling the threads on the same core typically results in a higher performance. This is because of lower thread management overheads as well as lower inter-thread communication costs.

Another presumably more important issue studied in the paper addresses the high implementation cost of TLS on MCAs. To support TLS on an MCA, the cache subsystem has to keep track of a large number of speculative versions of the same memory block. To track the correct version and to detect misspeculations, each cache block must bookkeep the thread identity along with additional status bits. In addition, the protocol to orchestrate the correct actions is complex. By allowing only a single speculative thread, the protocol can be simplified significantly and the cache overhead can be reduced. The question is how much performance is lost? Our main contribution is the design principles and performance evaluation of the *dual-thread speculation architecture*. Our detailed evaluation of this design point shows that it can exploit most of the thread-level par-

*The authors are members of the HiPEAC Network of Excellence funded by the EU under IST FP6.

allelism of TLS on an MCA but with dramatically lower design complexity. Hence, it follows the spirit of the saying: *"A bird in the hand is worth more than two in the bush."*

In Section 2, we present the baseline system and the support mechanisms and protocols needed for TLS and frame the problems addressed in the paper. We then present the simplified TLS support for the dual-thread speculation architecture in Section 3. In Section 4, we discuss the evaluation methodology followed by the experimental results in Section 5. Related work is discussed in Section 6 followed by our conclusions.

2 Architectural Framework

2.1 Baseline Multithreaded Chip Architecture

The baseline MCA architecture is shown in Figure 1. It consists of a number of processor cores, each having a private level one (L1) data and instruction cache, which are attached via a bus interconnect to a shared level two (L2) cache. A snoopy-cache, write-invalidate protocol maintains consistency among the L1 caches.

Each core supports simultaneous multithreading, which allows several of the microarchitectural resources to be shared between threads, e.g., the L1 caches, the physical registers (registers) and the execution units (EUs). Conversely, some resources have to be replicated such as fetch mechanisms including a program counter for each thread and the load/store queues as shown in Figure 1.

The assumed TLS system extracts threads from two sources: program loops and procedures/methods. In the former case, when a loop is detected, consecutive iterations are spawned as speculative threads. In the latter case, when a procedure call is encountered, a speculative thread executes the continuation of the code, after the procedure call, referred to as *module-level speculation*. In both cases, sequential semantics is respected; the execution result is the

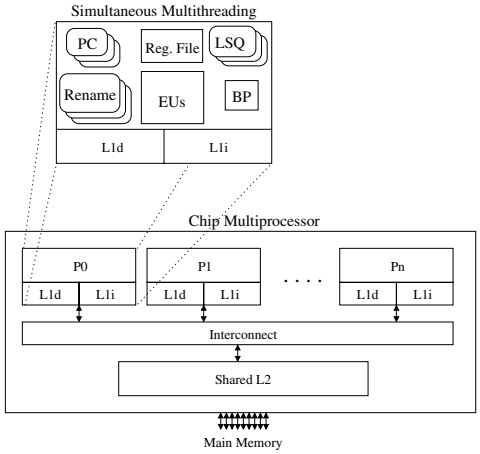


Figure 1. Baseline architecture.

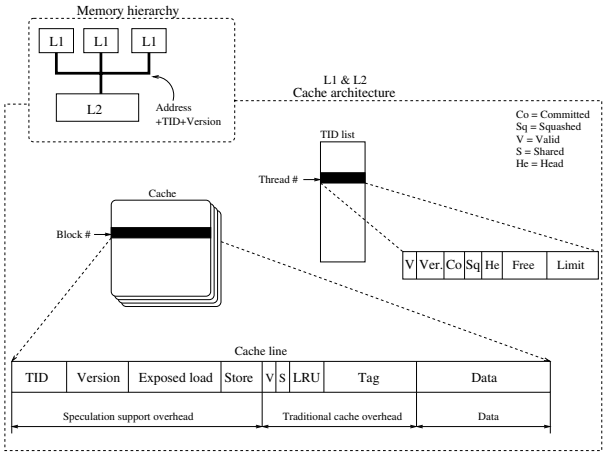


Figure 2. Speculation support in caches.

same as if the threads were executed sequentially in program order. Therefore, speculative threads are enumerated so as to reflect their position in the program order.

To maintain sequential semantics, dependences between threads are respected. All locations read by a speculative thread `i` are marked. If a less speculative thread, that is a thread with a lower number (in program order) $j < i$, subsequently modifies such a location, a data dependence violation is triggered. If that happens, thread `i` and all more speculative threads, i.e. threads with a higher number in program order, have to be rolled back and re-executed. To avoid roll-backs on name dependency violations, a new version of a location is created every time a speculative thread stores a value for the first time. Because multiple versions of a location can exist, the correct version has to be located; a read must return the value from the version associated with the most recent thread in program order that modified it.

To detect data dependence violations and to manage all versions, the snoopy-cache protocol in the baseline architecture is extended as shown in Figure 2. Apart from the state information associated with each block to implement an MSI snoopy-cache protocol, [1] the extra state information needed to support TLS is depicted in the figure. Each block is extended with two bits per word – called *exposed load* and *store* bits – designating whether a speculative thread has read or written to that word, respectively. The exposed load bit is set if and only if a speculative thread loads that location and the store bit is not set.

A new thread is assigned a thread identity (TID) to position it in program order. The child thread is always more speculative than the parent and inherits the order of the parent with respect to all other threads. Therefore, the new thread is assigned a TID higher than the parent, but lower than all more speculative threads in the system. For each TID, there is an entry in the TID list shown in Figure 2. The

[1] State bits V and S encode INVALID, SHARED, and MODIFIED.

TID list stores *free*, which is the largest TID higher than the current that is not already in use. To allow for spawning threads out-of-order, i.e. spawning new threads which have both more and less speculative threads in the system, the new thread is assigned a TID in between the spawning threads TID and *free*, thus leaving gaps in the TID sequence for future threads. When a speculative thread accesses a cache block for the first time, the block is tagged with its TID. This makes it possible to separate between multiple versions of the same cache line owned by different threads.

When a thread writes to a block and that block is in the SHARED state, the write is broadcast and the snoopy-cache protocol will ascertain that all other copies in the L1 and the L2 caches are queried. The other caches have to lookup the address, and if another version (or possibly several other versions) of the location is found, compare the TID with the TID of the broadcast write. If a version with a higher TID is found and the exposed load bit is set for the location being written, a data dependency violation is triggered.

Likewise, when a thread reads from a block that is not present in its L1 cache, the snoopy-cache protocol will broadcast the read request to the other caches. Each cache can do a local TID comparison to find its most recent version of the requested block. However, to find the globally most recent copy in the system, centralized version control logic comparing the most recent versions from each cache is needed. We assume version control logic similar to what is described by Gopal et al. [3].

If the thread succeeds, its speculative state is committed by setting the *commit* bit in the TID list. The committed cache blocks can be written back lazily as with a normal write-back protocol. However, every now and then the cache has to be sweeped to make sure there are no blocks left with committed TIDs, so that the TIDs can be reused.[2] The non-speculative, or head thread, has the *head* bit set in the TID list, and is the only thread allowed to commit. When it commits, the *head* bit is set for the next higher used TID. If a thread misspeculates, the *version* field in the TID list is incremented and the thread is restarted. The version is also recorded in each cache block touched by the thread, and makes it possible to avoid using out-of-date speculative values after the restart, that is, all speculative values can be squashed by incrementing *version*. Alternatively, the *squashed* bit is set if the thread should not restart.

Since the TID list keeps track of which threads are still running, restarted, squashed, or committed, all version comparisons must be accompanied with a lookup in the TID list. A block from an out-of-date version of a restarted thread should not trigger a violation if a less speculative thread writes to the block, and reads should not return versions created by a squashed thread.

[2]Since TIDs are reused the sequence sometimes wraps around, which must be taken into account when determining the order of threads.

Table 1. Tradeoffs between running threads on the same versus different cores.

Actions	*Same core*	*Different core*
Thread start, restart	Registers need not be copied through memory.	Inter-core communication.
Data sharing	Through L1 cache on same core.	Inter-core communication.
L1 cache access	Shared between the threads.	L1 cache resources scale with the number of cores.
Exec. unit access	Shared between the threads.	Number of EUs scale with the number of cores.

This protocol works much like the one described by Renau et al. [12]. There are simpler protocols, such as the speculative versioning cache (SVC) [3]. SVC uses gang commit and squash, which means all cache lines in an L1 are committed or squashed in one operation. However, to support SMT, a protocol which allows mixing versions from different threads in the same cache is necessary, which precludes gang operations. In addition, out-of-order spawn adds some complexity, but is important for performance reasons when exploiting module-level parallelism; this has been demonstrated by Renau et al. [13].

2.2 Discussion

This paper focuses on two issues associated with the management of speculative threads in the baseline system: (1) where to start a new thread – on the same core as the parent thread or on another core? (2) how the complexity of the version management can be reduced by only allowing a single speculative thread.

Starting with the first issue, the performance overheads of running threads on the same core versus on different cores differ in two respects. Obviously, inter-thread communication cost is higher between threads on different cores, which translates into performance losses. However, it is a scalable approach where the threads do not contend for local resources. By contrast, the resources on a single core are limited and the threads have to share them. This is especially a concern for the L1 cache and the execution units that can be overcommitted by scheduling multiple threads on the same core. In Table 1, we compile the major differences between running the threads on the same versus different cores.

When spawning a new thread the input register values and initial program counter must be transferred from the original thread to the new thread. This can be handled quickly within an SMT processor [1, 11, 9], but to start a new thread on another core the data must be transferred

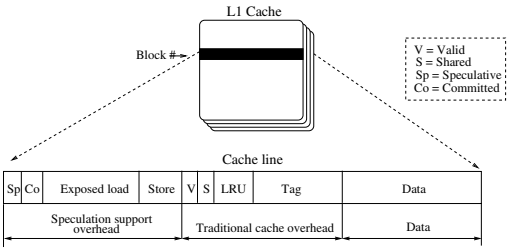


Figure 3. Dual-thread speculation support.

across the on-chip bus. When a thread commits, the speculation system must check for register dependences by comparing the final register contents of the committing thread with the inputs of its successor. This also imposes some overhead. Therefore, thread management is less efficient for threads on different cores.

In summary, it is an open question whether one should spawn off a new thread on the same core or another and we focus on this issue quantitatively in Section 5.

As for the complexity of the TLS support in the baseline system, an advantage of the version management scheme assumed is that a speculatively modified block is allowed to be evicted from the L1 cache without causing the thread to stall. Additionally, threads can migrate between cores and the scheme can still locate all versions correctly.

We note however that managing multiple speculative threads adds to the complexity of a plain snoopy-cache protocol in two respects. First, each block in the L1 and L2 caches is extended with status bits and TIDs. To avoid running out of TIDs in the gaps left to support out-of-order spawn, the TID field needs to be sufficiently large. We have used a 32-bit TID and an 8-bit version number in our simulations. Factoring in the status bits, 56 extra bits are needed per block, which yields 22% overhead assuming a 32-byte cache block. Even if implementations with less overhead are possible [13], the overhead is still noticeable. Second, the basic snoopy-cache protocol has to be extended to be able to locate the correct version upon a cache miss and detect data dependence violations, which adds to its complexity. We next study a design alternative by which the complexity of the TLS system can be reduced significantly.

3 The Dual-Thread Speculation Architecture

With only one speculative thread, there is no need to keep track of which threads are more or less speculative or retain the possibility to spawn threads out-of-order. Therefore, the TID can be replaced with a single *speculative* bit indicating if the thread is speculative or not. Figure 3 shows a cache block for dual-thread speculation.

The reason for keeping the *commit* and *squash* bits in the TID list is to allow versions from several threads sharing the same cache. With only one speculative thread, gang commit or squash [3], i.e. invalidating or committing all speculative values in the cache can be used. To this end, a *commit* bit is associated with each cache line. A thread commit is performed by setting all commit bits in the cache, and a squash by clearing all *valid* bits for blocks with the *speculative* bit set. Commit bits are cleared when the block is reused by a new speculative thread. The cache overhead for this scheme is 18 bits or 7% with a 32-byte cache block.

Protocol transactions are also simplified. Writes from the non-speculative thread causes a squash if the speculative thread has a version with the exposed load bit set. Reads from the speculative thread request a copy like in a traditional memory hierarchy, if there is not already a speculative version. No TID comparisons are necessary to locate one out of many speculative versions.

With these simplifications to the protocol, there is no longer a need for maintaining a TID list or a mechanism for periodically sweeping the cache to free up TIDs. If we restrict the dual-thread system to running the speculative thread on the same core as the non-speculative thread, the mechanism for transferring the thread's initial values to another core via the on-chip bus can be eliminated as well. Finally, we can eliminate the overhead in the L2 cache by eliminating the possibility to evict speculative values from the L1. This will reduce the available space for speculative state, but on the other hand, with only one speculative thread less space will be needed. There can be no more than two versions of a location at a time.

4 Simulation Methodology

We simulate the baseline system in the previous section with the architectural parameters summarized in Table 2. The machine parameters are chosen to resemble a modern out-of-order processor. Thread start, restart, and commit overheads for SMT processors are lower due to the possibility to implement faster in-core mechanisms. The bus transfers for remote-core threads are needed to copy register values between threads at start, restart, and for register dependence checking at commit.

The benchmarks are primarily from the SPEC CPU benchmark suites. Art and Equake are from CFP2000 (floating point benchmarks), both memory-intensive. Gzip, Perlbmk, Vortex, and Vpr are from CINT2000 (integer benchmarks). M88ksim is from CINT95, Neuralnet (nnet) from jBYTEmark (java benchmark) and Deltablue (dblue) is an application from Sun. M88ksim benefits the most from ILP of all programs. While Art and Eqake have mostly loop-parallelism, the others contain a mix of loop- and module-level threads with TLS. The benchmarks are picked to both represent different behavior and to allow for comparisons to other TLS papers. Some other CINT2000 benchmarks were tested but not found to gain any performance from TLS, and therefore not included. The applications and input sets are listed in Table 3.

Table 2. Baseline machine parameters.

Feature	*Parameters*		
	2-issue	*4-issue*	*8-issue*
Processor parameters			
Fetch/issue/commit width	2/2/2	4/4/4	8/8/8
ALUs per core	2	4	8
Load/Store units per core	1	2	3
Floating point units per core	1	2	3
Instruction window size	128	256	384
LSQ size	48	64	96
Branch pred. per cycle	2	2	2
Pipeline length – integer	8	8	8
Pipeline length – float. pt.	11	11	11
Branch predictor	G-share, 16k pred. table, 8-bit global history, 2k target buffer, 8 entry RAS/thread.		
L1 caches	32+32 kB (i+d), 4-way ass. 3-cycle load-to-use latency 3 cycles for version upgrade or new version block copy.		
L1 instruction cache	Sequential prefetching		
Shared L2 cache	4 MB 32-way ass. 15-cycle total latency without contention		
Cache block size	32 bytes		
On-chip bus	256 bits, 2 cycle latency		
Main memory	200 cycle latency		
Thread-management Overheads			
Same-core thread-start	10 cycles (no bus activity)		
Same-core restart	5 cycles (no bus activity)		
Same-core commit	5 cycles (no bus activity)		
Remote-core thread-start	20 cycles + 4 bus transactions		
Remote-core restart	10 cycle + 1 bus transaction		
Remote-core commit	10 cycles + 4 bus transactions		

The benchmarks are first run sequentially on Simics 1.8 [8], a full-system instruction-set simulator which mimics a SPARC workstation with all necessary devices and runs the Solaris operating system. The applications are compiled with the Sun Workshop 6 compiler with aggressive optimizations. From Simics, an instruction trace is generated. The instruction trace is used to feed our trace-driven TLS simulator. Since the benchmarks typically run for billions of instructions, we have not been able to simulate the entire run. Instead, four samples, each containing 25 million instructions, have been taken from each benchmark.[3]

5 Experimental Results

5.1 Performance of Baseline CMP

Figure 4 shows results for an 8-way CMP with 4-issue processors and a single thread per core. For each application there are four bars, each showing speedup compared to executing it on a single 4-issue processor. The leftmost bar shows the sequential execution, the remaining three show speedup with only loop-level speculation, only module-level speculation, and both forms combined. The legends for each bar in this and the following graphs indicate the number of threads (8t), issue-width (4i), and type of parallelism (loop, mods, or both). The speedup ranges from nothing up to 1.6 for Art. Speedup is both due to prefetching, i.e. when speculative threads that misspeculate have fetched cache blocks which can be used by subsequent threads, and due to actual parallel execution. The prefetching effect is most important in memory-intensive programs like Equake and Vpr.

Run-length prediction [17] reduces the number of short threads, which typically do more harm than good for TLS. It has been used for CMP models with at least 4 threads with improved results, but not for the SMT models or 2-thread CMP where it sometimes hurt performance instead.

[3]Except Art where 8 samples were used since the variation was larger.

Table 3. Benchmark applications.

App	art	dblue	equake	gzip	m88k
Input	ref.110	default	ref	ref.log	ref
App	nnet	perlbmk	vortex	vpr	
Input	default	ref.perf	ref.1	ref.route	

5.2 SMT vs. CMP

Figure 5(a) presents the results of a comparison of two designs that both support four threads. One is an 8-issue SMT, and the other a CMP with four 2-issue cores. That is, in total they can issue the same number of instructions per cycle. The speedup in this figure is computed relative to sequential execution on a single 8-issue processor.

Overall, the SMT machine, labelled with *smt-4t* in the figure legends, performs somewhat better. This is true both for memory-bound applications such as Art, and high-ILP applications such as M88ksim. When the SMT processor is not throttled by lower total fetch- or issue width, cache or resource contention does not seem to slow down the SMT significantly – the SMT benefits from its lower overheads. Simulations with a 2-thread 4-issue SMT processor compared to a 2-way 2-issue CMP were also conducted. The results are similar, though the differences between the machines are smaller.

In addition to the baseline parameters, simulations were conducted with both the total size and the associativity of the level one caches on the SMT processor scaled with the number of threads. While the baseline machine uses 32 kbyte 4-way level one caches, the 4-thread SMT was equipped with 128 kbyte 16-way caches. While building a 16-way L1 cache might not be a realistic design point, this experiment provides a comparison of the two architectures that is not influenced by differences in cache space. The results are shown in Figure 5(b) – the speedup for Neural-

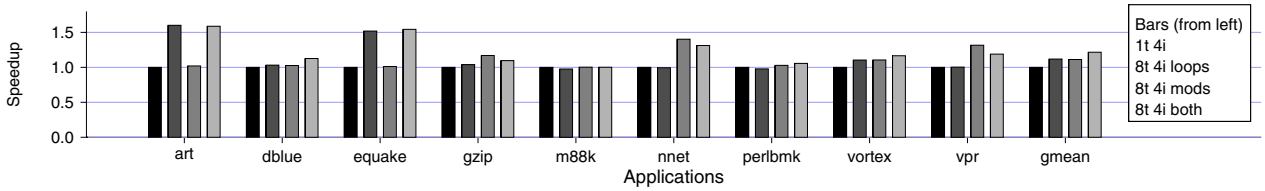
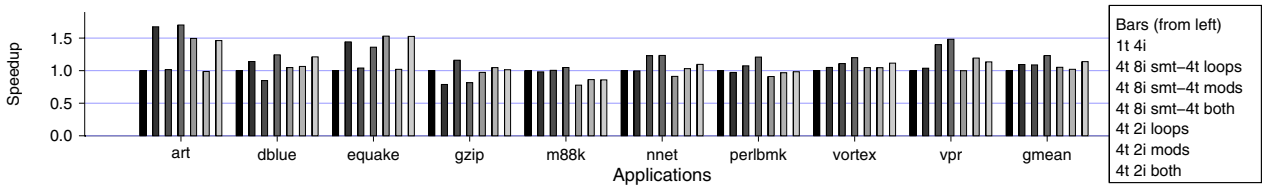
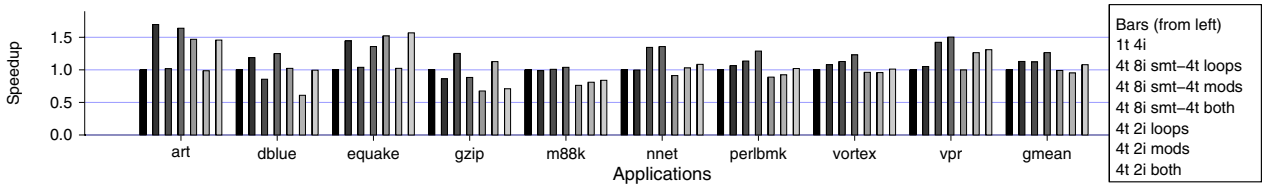


Figure 4. TLS on 8-way CMP with 4-issue processor cores.



(a) *4-thread 8-issue SMT compared to a 4-way 2-issue CMP.*



(b) *Speedup where the SMT has the same total L1 cache size as the CMP.*

Figure 5. Comparing SMT and CMP designs with equal total issue width.

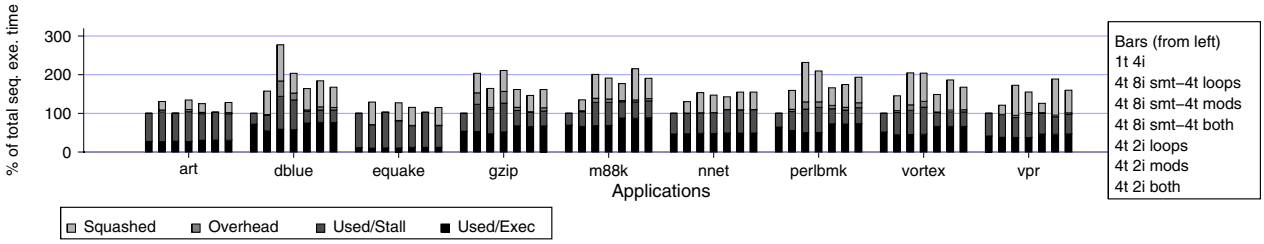
net and Perlbmk sees a healthy increase, and several other applications execute somewhat faster. Equake is the only application where the CMP performs slightly better, it already has quite long threads and little overhead. Therefore, it does not gain much from the lower thread management overhead. In addition, even though Equake is memory-intensive, it does not benefit from more L1 cache space.

Figure 6(a) shows the total execution time, i.e. the sum of the execution times for all executed threads as a percentage of the execution time for the sequential execution. As we can see the overhead is quite substantial in some cases. The reason why the total execution time for the SMT is sometimes longer than for the CMP is the use of run-length prediction, and therefore fewer speculative threads, for the CMP. The *Used/Exec* and *Used/Stall* segments of the bars show the fraction of cycles where committed threads issued instructions or stalled, respectively. Inspection of the execution time breakdown reveals that the SMT version of Equake has somewhat more stall time. This is not due to memory stall, in fact the total memory stall is slightly higher for the CMP version. However, the resource sharing in the SMT seems to introduce some stall time in the pipeline.

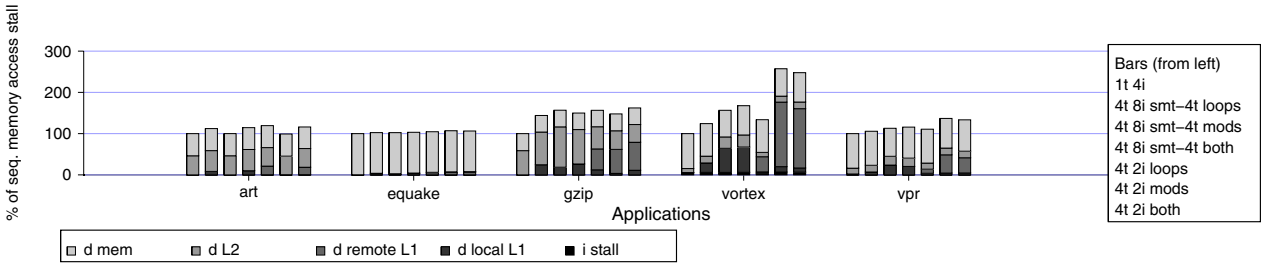
Figure 6(a) also shows that, as expected, the thread management overhead is in general lower for the SMT machine, while the execution overhead from squashed threads is approximately the same. Experiments show that even if the issue-width is doubled for the cores in the CMP, the results do not change much. The performance advantage for the SMT processors is due to the lower overhead, which the CMP cannot make up even with wider-issue processors.

Figure 6(b) shows memory stall times (only some applications are included to save space). The total stall for all threads is compared to the stall in the sequential execution. For many applications, the remote L1 stall due to inter-thread communication is significant with the chip multiprocessor. Naturally, this overhead does not exist for the SMT. However, for a few applications like Vortex the local L1 stall does have a performance impact. Local L1 stall for the SMT is due to updating block versions or duplicating cache blocks from other threads, i.e. inter-thread communication handled locally. This overhead partly replaces the overhead due to remote L1 stall seen in the CMP models. Gzip and Neuralnet also suffer from a larger fraction of L2 misses for SMT. They are two of the applications which benefit the most from a larger L1 cache in the SMT simulations.

In summary, the lower thread management overhead and reduction in remote L1 stall originating from thread communication makes an SMT more efficient for thread-level



(a) *Execution time breakdown.*



(b) *Data stall time breakdown.*

Figure 6. Execution statistics for a 4-thread, 8-issue SMT compared to a 4-way 2-issue CMP.

speculation compared to a CMP with the same total issue width. However, for some applications the performance gain can be limited unless the SMT has a large enough L1 cache. In addition, the 4-thread SMT performs on par with the 8-way 4-issue CMP.

5.3 Performance with a Single Speculative Thread

The performance of a machine with one speculative thread is evaluated both for an SMT, where both threads run on the same core, and on a 2-way CMP with one thread on each core. The results in Figure 7(a), are for the SMT. The SMT is a 4-issue processor with support for two threads.

Performance-wise, the speedup for Vpr is about half that of the baseline 8-way CMP. For Gzip, Neuralnet and Art more than half the speedup, and for Equake almost all the available speedup is still exploited with this simple machine. Figure 7(b) shows the same experiments for a 2-way CMP, with 4-issue cores. The performance for the CMP machine is slightly lower than for the SMT, but even with this machine model much of the parallelism can be exploited with one speculative thread.

This experiment shows that compared to results with the best 8-way CMP or 4-way SMT models, a reasonable amount of the available parallelism can be exploited with only one thread. With this in mind, it seems to be an interesting design point given the possible reductions in hardware complexity.

6 Related Work

DMT [1] and IMT [11] are TLS architectures for SMTs. Both manage the speculative threads completely within the SMT core. Therefore, thread size should typically be small. Marcuello and González [9] propose an SMT architecture specialized on loop parallelization. Speculative threads share the fetch bandwidth when several loop iterations execute along the same control path. However, none of these works explicitly compare the the advantages and disadvantages of an SMT with a comparable CMP architecture.

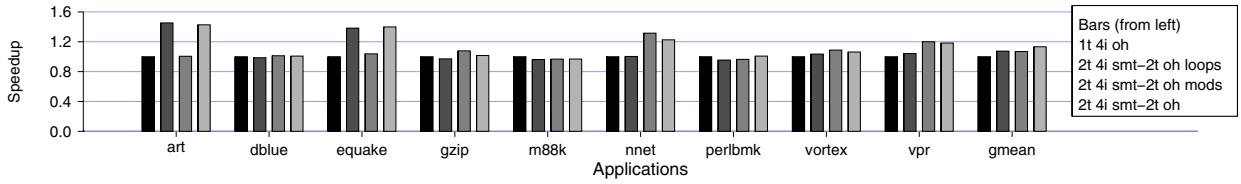
7 Conclusions

We have investigated the performance of TLS on SMT processors. It was found that in general, TLS performs better on an SMT given equal fetch and issue capacity as a CMP. This is due to lower thread management overheads and reduced inter-thread communication costs. For some applications, however, it is necessary to scale the level one cache with the number of threads in order to achieve optimal performance on the SMT.

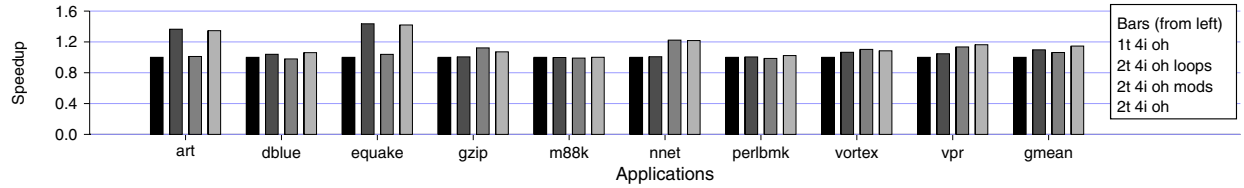
The experiments with a one speculative thread showed that, somewhat surprisingly, most of the parallelism exploited by the 8-way CMP could also be exploited with this simple model. While the scalability is limited, the lower complexity makes it an interesting design point.

Acknowledgements

This research has been funded in part by the Swedish Research Council and in part by the SARC Integrated Project



(a) *Results with 2-thread 4-issue SMT processor.*



(b) *Results with 2-way 4-issue CMP.*

Figure 7. TLS with a single speculative thread.

funded by the EU under the FET programme. The computing resources made available by the SNIC programme are also appreciated.

References

[1] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proc. of the 31st Annual International Symposium on Microarchitecture (MICRO '98)*, pages 226–236. IEEE Computer Society, Dec. 1998.

[2] L. Codrescu and D. S. Wills. Architecture of the atlas chip-multiprocessor: Dynamically parallelizing irregular applications. In *Proc. of the 1999 International Conference on Computer Design (ICCD '99)*, pages 428–435. IEEE Computer Society, Oct. 1999.

[3] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative versioning cache. In *Proc. of the Fourth International Symposium on High-Performance Computer Architecture (HPCA '98)*, pages 195–206. IEEE Computer Society, Feb. 1998.

[4] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proc. of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII '98)*, pages 58–69. ACM Press, Oct. 1998.

[5] J. Kalla, B. Sinharoy, and J. Tendler. IBM power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[6] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[7] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.

[8] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation plattform. *IEEE Computer*, pages 50–58, Feb. 2002.

[9] P. Marcuello and A. González. Exploiting speculative thread-level parallelism on a SMT processor. In *Proc. of the International Conference on High Performance Computing and Networking (HPCN '99)*, pages 754–763, Apr. 1999.

[10] T. Ohsawa, M. Takagi, S. Kawahara, and S. Matsushita. Pinot: Speculative multi-threading processor architecture exploiting parallelism over a wide range of granularities. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture (MICRO '05)*, pages 81–92, Washington, DC, USA, 2005. IEEE Computer Society.

[11] I. Park and T. N. Vijaykumar. Implicitly-multithreaded processors. In *Proc. of the 30th Annual International Symposium on Computer Architecture (ISCA '03)*, pages 39–50. IEEE Computer Society, June 2003.

[12] J. Renau, K. Strauss, L. Ceze, W. Liu, S. Sarangi, J. Tuck, and J. Torrellas. Thread-level speculation on a cmp can be energy efficient. In *Proc. of the International Conference on Supercomputing (ICS '05)*. ACM, June 2005.

[13] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas. Tasking with out-of-order spawn in tls chip multiprocessors: Microarchitecture and compilation. In *Proc. of the International Conference on Supercomputing (ICS '05)*. ACM, June 2005.

[14] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pages 414–425. ACM Press, June 1995.

[15] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proc. of the Fourth International Symposium on High-Performance Computer Architecture (HPCA '98)*, pages 2–13. IEEE Computer Society, Feb. 1998.

[16] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. of the 22th Annual International Symposium on Computer Architecture (ISCA '95)*, pages 392–403. ACM Press, June 1995.

[17] F. Warg and P. Stenstrom. Improving speculative thread-level parallelism through module run-length prediction. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS '03)*. IEEE Computer Society, Apr. 2003.