Supporting Speculative Multithreading on Simultaneous Multithreaded Processors

Venkatesan Packirisamy, Shengyue Wang, Antonia Zhai, Wei-Chung Hsu, and Pen-Chung Yew

Department of Computer Science,
University of Minnesota, Minneapolis
{packve, shengyue, zhai, hsu, yew}@cs.umn.edu

Abstract. Speculative multithreading is a technique that has been used to improve single thread performance. Speculative multithreading architectures for Chip multiprocessors (CMPs) have been extensively studied. But there have been relatively few studies on the design of *speculative* multithreading for *simultaneous multithreading* (SMT) processors. The current SMT based designs - IMT [9] and DMT [2] use load/store queue (LSQ) to perform dependence checking. Since the size of the LSQ is limited, this design is suitable only for small threads. In this paper we present a novel cache-based architecture support for *speculative simultaneous multithreading* which can efficiently handle larger threads. In our architecture, the associativity in the cache is used to buffer speculative values. Our 4-thread architecture can achieve about 15% speedup when compared to the equivalent superscalar processors and about 3% speedup on the average over the LSQ-based architectures, however, with a less complex hardware. Also our scheme can perform 14% better than the LSQ-based scheme for larger threads.

1 Introduction

With increasing amount of resources available for the processor, architects are going for multithreading-based designs like CMPs and SMTs. At present, these architectures are mainly used to improve the processor throughput. Using these multithreaded architectures to improve single thread performance still poses a challenge. Speculative multithreading [5, 11] is one way to utilize the multiple threads to improve single thread performance. Here, threads are automatically extracted from a sequential program by a compiler and executed in parallel to improve its execution time. Architecture support is needed to detect any dependence violation, and also to buffer the results created by speculatively created threads.

Existing SMT based speculative multithreading approaches either use complex hardware [7] or use limited resources like LSQs [9, 2] to buffer speculative results, and to record load addresses to check for dependence violations. The advantage of LSQ-based method is that the LSQs are already available to the processor, so the technique does not need any major modifications to the

Y. Roberts et al. (Eds.): HiPC 2006, LNCS 4297, pp. 148–158, 2006.
© Springer-Verlag Berlin Heidelberg 2006

processor architecture as in the case of [7]. Also, in the LSQ, entries are created for each load and store operations, so the dependence checking granularity is at the byte level. At the byte level, there could be no false dependences. But the LSQ entries for speculative threads are not cleared till the thread commits. So the main disadvantage in using LSQs is their limited size since it is not cost effective (or power efficient) to have large LSQs. Due to this consideration, LSQ based architectures can support only small threads. But our research [14] shows that if we need to consider a more realistic overhead of forking a thread, it becomes more difficult to justify at small granularities. Hence, it is important to support larger threads.

In this paper, we propose a novel cache-based architecture to implement speculative multithreading in SMT processors that only requires a few extra bits to each cache line in existing L1 cache in SMT. Also our approach can handle large threads since now the entire cache can be used to buffer results and to check for dependences.

The rest of the paper is organized as follows: section 2 discusses related work, section 3 discusses our cache-based architecture to support speculative multithreading, section 4 discusses results and in section 5 we conclude the paper.

2 Related Work

Speculative multithreading architectures have been studied intensely during the past decade. Earlier architectures were based on special hardware structures for dependence checking like the *address resolution buffer (ARB)* in [4], and the *memory disambiguation table (MDT)* in [5]. These special hardware structures are of limited size and need extra cycles to access them. To avoid these limitations cache-based architectures like *speculative versioning cache (SVC)* [13] and *STAMPede*[10] were proposed.

When compared to speculative multithreading on *chip multiprocessors (CMPs)*, there are very few studies on supporting *speculative multithreading* for SMTs. In [7], private L1 cache for each context is used to buffer speculative values and do dependence checking. In DMT [2] and in IMT [9] an enhanced LSQ is used.

The main limitation of the LSQ-based approach is the limited size of the queue. To overcome this limitation we propose a cache-based scheme in this paper. We draw many ideas from the cache architectures proposed for CMPs. The difference is that the CMP-based architectures have private L1 cache for each core and is used to buffer results. The dependence checking hardware is also distributed among different L1 caches. In our approach, all the contexts in the SMT share the same cache.

Concurrent to our work, Stampede [11] has extended the cache protocol described in [10], to support shared cache architectures. Their technique was studied in the context of multi-core processors using shared cache. In [3], shared L2 cache based technique was used to speculatively parallelize database applications. Though they mention that it could be applied to SMT processors all their results and conclusions are for CMPs, while our scheme is specifically aimed at

SMT processors. Also we propose a novel two-thread scheme and our four-thread scheme uses fewer bits per cache line than their scheme. A detailed comparison of our technique with the STAMPede technique is beyond the scope of this paper.

3 Speculative Simultaneous Multithreading

3.1 Basic SMT Architecture

We consider a SMT architecture where many resources like fetch queue and issue queue are fully shared [12]. Fig. 1 gives a block diagram of the SMT architecture. To implement speculative multithreading, we need hardware support to buffer results from speculative threads, detect dependence violation between threads, and synchronize threads to communicate register values. The only modifications we need are the signal table and the modified L1 data cache. Our inter-thread register synchronization scheme is very similar to [15].

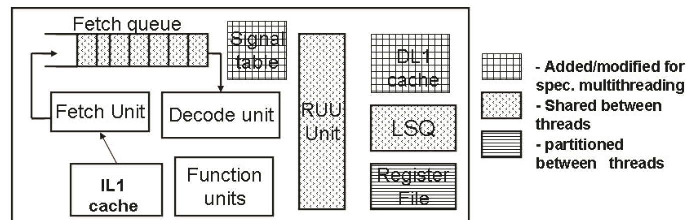


Fig. 1. SMT Block Diagram

We use a novel cache-based scheme to support buffering of speculative values and to enforce memory dependences. In section 3.2, we first present a simplified scheme that supports only one speculative thread, and in section 3.3 we extend this scheme to four (or more) threads.

3.2 Simplified Two-Thread Scheme

In this section, we consider a SMT processor with only two threads. Here, we only need to introduce two extra states to each cache line - *Speculative Valid (SV) and Speculative Dirty (SD)*. Each cache line also needs two extra bits - *Speculative Load (SL) and Speculative Modified (SM)* to support data dependence checking. In the proposed scheme, all of speculative data are kept only in the shared L1 cache, and all of the data stored in L2 cache are non-speculative. Fig. 2 presents the cache-line state transitions in this scheme. In fig. 2 the transitions are of the form 'Command from processor / Action taken'.

Speculative value buffering. When a speculative thread writes, the value is stored in the shared L1 data cache with the SM bit of the cache line set and the cache line transitions to the SD state. The value stays in the cache till the thread is committed or squashed. Thus, the L1 D-cache acts as a *store buffer* that buffers speculative updates.

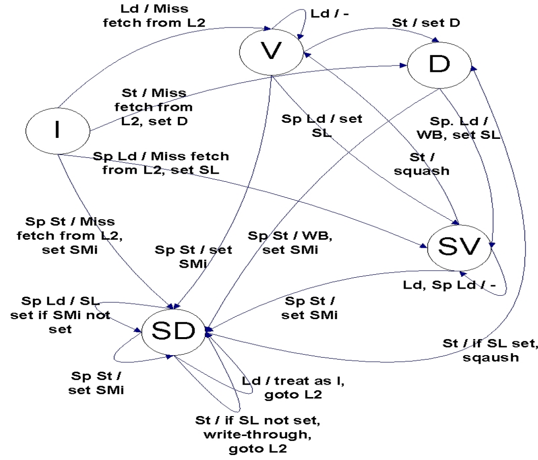


Fig. 2. Two Thread Scheme - Cache State Transitions

Dependence Violation Detection. When a speculative thread issues a load operation, it first checks if a speculative thread has already written the value. However, by having just one SM (speculative modified) bit for each cache line, we cannot be sure which word in a particular cache line was written by the speculative thread. To allow more precise dependence information, we could maintain one SM bit (SMi) for each word in the cache line. If the SMi bit is not set, the SL (speculative load) bit will be set and the cache line transitions to SV (speculative valid) state, as this load could cause a possible dependence violation, when a non-speculative write arrives later.

Here, when a non-speculative thread writes into a cache line, if the SL bit is already set, it indicates that the speculative thread has read a stale value. The speculative thread will be squashed and restarted.

Non-speculative thread execution. If the state of the cache line being written to is SD (speculatively dirty), the non-speculative thread writes the value directly to L2 cache. Also, it writes the portion of the data non-overlapped with the speculatively modified data (indicated by SMi bits) into the L1 cache. This merging is done, so that the speculative thread can get the most recent non-speculative value from L1 cache. Also this simplifies the commit operation. Reads by a non-speculative thread to a speculatively modified line (SD) are treated as a *cache miss* and the value is directly taken from the L2 cache.

Replacement policy. Speculatively modified cache lines or the lines with the SL bit set cannot be evicted from the cache. If evicted, we lose information which can lead to incorrect execution. When we have to replace a line, a line which has none of the SL and SM bits set is selected. If a non-speculative thread needs to replace a line and couldn't find a clean line, the speculative thread is squashed to relinquish its lines. This is done to avoid blocking the non-speculative thread and

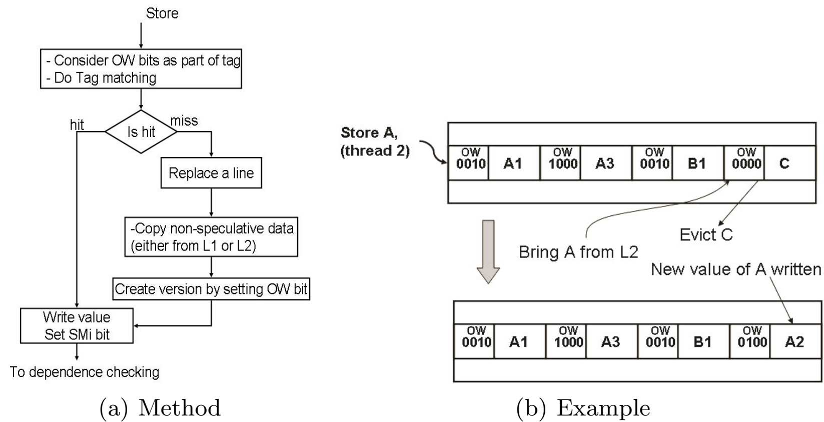
thus avoiding deadlock. In case of speculative thread, the thread is suspended till it becomes non-speculative.

Commit and Squash. When a thread commits, both the SL and SMi bits are cleared. Unlike other schemes where every speculative value needs to be written to the cache at the point of commit (which could potentially take hundreds of cycles), the commit operation can be done in just one cycle in our scheme by gang-clearing both SL and SMi bits.

When a thread squashes, the SL bit in all cache lines is cleared (gang-clear). The valid bit for a cache line is also cleared if the SM bit is set. This is like the conditional gang-clear operation used in Cherry[8].

3.3 Four-Thread Scheme

When executing more than one speculative thread, the L1 D-cache needs to buffer results from two or more threads, so the two-thread scheme cannot be directly applied. In this section we propose a scheme which can efficiently handle more than one speculative thread. The basic idea is to use the entire set in the cache to buffer different versions of the same line created by the different threads. We will use a 4-thread system to simplify our explanation.



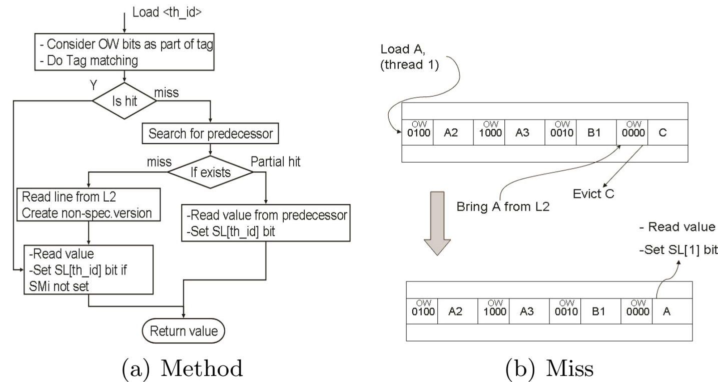
(a) Method (b) Example

Fig. 3. Speculative Store Handling

Speculative Buffering. The L1 D-cache has to buffer results from multiple threads, so we introduce *Owner* bits (OW) (one for each thread) which keep the speculative thread id that wrote into the cache line. For a non-speculative cache line, the OW bits are cleared. Buffering of speculative values is explained in Fig 3(a). Fig 3(b) shows an example where thread 2 tries to write a new version of A to a set which already contain versions from thread 1 and thread 3.

Speculative Load Execution. A cache line can be read by any of the four threads, so a single SL bit is not sufficient to indicate which thread has caused dependence

violation. We introduce a SL bit for each thread on each line of cache (4 bits for 4 threads). The execution of a *speculative load* instruction is explained in fig 4. We can see that the *speculative load* can either load from its own version (i.e., a hit), from predecessor thread's version (i.e., a partial hit) and from L2 cache (i.e., miss - fig. 4(b)). of the cache line.



(a) Method (b) Miss

Fig. 4. Speculative Load Handling Example

Dependence Detection. When a store executes, it checks whether the versions of the cache line belong to any of its own successor threads. If SL bit is set for any of the successor threads, the successor thread is squashed along with its successors. The oldest squashed thread is then restarted.

Non-Speculative Thread Execution. Execution of non-speculative load and store is very similar to the speculative thread execution. But the non-speculative thread does not set any SL, OW or SMi bits. Also, the non-speculative store writes the portion of the data non-overlapped with speculatively modified data (SMi bits) into all versions in the L1 cache. This merging is done so that the speculative threads will get the most recent non-speculative version.

Commit and Squash. To squash a thread, the SL[thread_id] is cleared for all of the lines in the cache. This can be done as a gang-clear operation. Also the line is invalidated if any of the SMi bit is set. This is accomplished by a conditional gang-clear operation as in two-thread scheme.

To commit a thread, the SL [thread_id] bit and the SMi bits of the thread are cleared. The commit must ensure two things. It should make sure that there is only one non-speculative version present in L1 cache. If a cache line to which the current thread wrote has another version which is earlier than that of the current thread, then that version needs to be written back and invalidated. Also, in our scheme, we require that only the non-speculative thread can send values to the successor threads. So once a thread becomes non-speculative, it has to send

its speculatively modified values to all successor threads. To ensure these two conditions are met, we need to commit each cache line belonging to the current thread whose SMi bit is set. To commit a cache line, we write-back and invalidate any versions that belong to earlier threads, and we need to merge the modified data with the versions belonging to successor threads. Though this step might be time consuming, we can see that this is simple to implement, and we can potentially overlap this with the execution of the next thread. Our simulation shows that this overhead causes no potential performance degradation.

Implementation Issues. While executing a *speculative load*, we may have to search the entire set in the cache to get the predecessor thread's cache line. Also, while detecting mis-speculation, we need to search the entire set to find if any successor thread has set the SL bit. These operations can be implemented by adding more logic to the tag matching hardware but it could increase cache hit time. In our scheme, we assume there is special hardware that does these "whole-set" operations, which is kept separate from the tag matching hardware. We assume such special operations take 3 cycles.

As we see in the two-thread case, we cannot replace a line with SL or SM bit set. If a speculative thread encounters a cache miss and if it is not able to find a clean line to replace from the cache, it can either suspend and wait till it becomes non-speculative or it can squash the successor threads and consume its cache lines. A thread will be forced to wait if it has no successors to squash. While waiting, a thread occupies shared resources like fetch queue, RUU and LSQ. There may be a situation where all the resources are occupied by the suspended thread and the non-speculation thread is unable to proceed, thus, causing a deadlock. To avoid this scenario, the speculative thread will give up its resources when it is stalled.

4 Experimental Results

4.1 Experimental Methodology

In our experiments, we used a detailed superscalar simulator based on Simplescalar 3.0. We modified Simplescalar to a trace based simulator that accepts Itanium traces. The trace for the input program is generated by Pin [6]. Traces are collected for four threads at a time and the simulator is called to consume the traces. Table 1 details the processor parameters used.

To generate parallel threads, we use a compiler framework based on Intel's ORC compiler [1]. The compiler selects loops in each benchmark that are suitable for parallel execution, performs optimizations such as code scheduling to enhance overlap between threads. The compiler also generates synchronization instructions for frequently occurred cross-iteration data dependences. Our compiler framework and the loop selection methodology are described in detail in [14].

Table 1. Processor Parameters

Fetch Width	4 Bundles (3 instructions each)
Decode, issue and commit width	8, 4 and 4 instructions
Function Units	4 integer, 4 floating point, 4 memory ports
Latency	1 cycle for integer, 12 cycle for floating point
Register Update Unit(ROB)	256 entries
LSQ size	128 entries
Branch predictor	Bimod, 2K entries
L1D,I Cache	64K, 4 way associative, 32B blocksize, 1 cycle
Unified L2	1MB, 8 way associative, 64B blocksize, 18 cycles
Memory latency	120 cycles for 1st chunk, 18 cycles subsequent chunks
Branch mis-prediction penalty	6 cycles

4.2 Results

Table 2 shows the details of benchmarks (from SPEC2000) used to evaluate our scheme. Since our primary objective here is to evaluate our proposed cache scheme when the SMT processor is executing in parallel mode, we only focus our simulations on the parallel regions in each benchmark.

Table 2. Details of Benchmarks

Benchmark	No of loops selected	coverage of selected regions
Mcf	6	60%
Twolf	15	32%
Vpr (place)	3	65%
Equake	4	90%
Art	12	52%

We consider the following configurations:

Superscalar: This is an out-of-order superscalar processor with parameters described in Table 1.
SMT-2: This is an out-of-order SMT processor which can support two threads at a time using the two-thread scheme described in section 3.2. This configuration has the same number of functional units as in the superscalar.Each line of cache has 9 extra bits (8 SMi and 1 SL).
SMT-4: This SMT processor can support four threads using the four-thread scheme described in section 3.3. It also has the same number of functional units as in (1) and (2). Each line of cache has 16 extra bits (8 SMi, 4 SL and 4 OW bits).
SMT-LSQ: This SMT processor supports 4 threads and uses the LSQ-based mechanism as in [9][2]. It has the same number of functional units, but uses extra space for enhanced LSQs that support speculation. Each thread has 128 LSQ entries.

CMP: This is a multi-core based speculative processor using the Stampede protocol[10]. This uses four identical cores and each core is a superscalar processor described in Table 1.

Fig. 5 shows the relative speedups of the different configurations over the superscalar configuration. From Fig. 5 we can see that the two-thread SMT scheme achieves about 10% speedup over the superscalar version. The two-thread SMT achieved this with very simple modifications to cache. The four-thread version achieved 15% speedup over superscalar. This performs better than the two-thread version but needed more complex hardware.

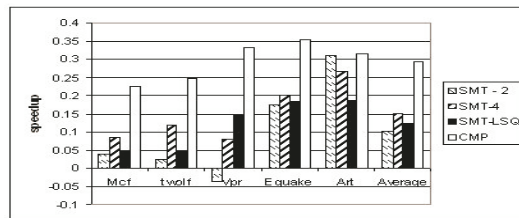


Fig. 5. Speedup of different configurations over the Superscalar configuration

Fig. 6 shows the execution time breakdown for the different configurations normalized to the execution time of the superscalar configuration. The explanation of the different categories are given in Table 3.

From Fig. 5, we can see that, usually the four thread configurations SMT-4 and SMT-LSQ perform better than SMT-2 configuration. This is because most loops selected by our compiler have good thread level parallelism and can benefit from more threads. However SMT-2 has the advantage of causing fewer squashes because it has only one speculative thread. Due to this SMT-2 performs better than SMT-4 and SMT-LSQ for the benchmark *Art*.

The CMP configuration performs about 15% better than the SMT-4 configuration. This is because the CMP uses four separate cores and, hence, has four

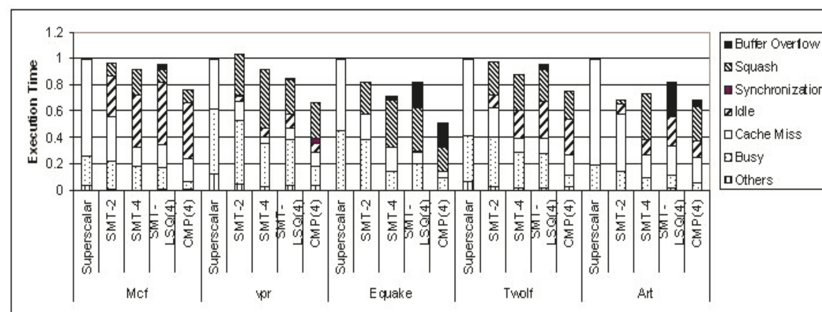


Fig. 6. Cycle breakdown normalized to the Superscalar configuration showing where the execution cycles are spent for different configuration

Table 3. Execution State Category

Category	Explanation
Cache Miss	Stall due to data cache miss
Idle	Lack of threads to execute
Synchronization	Thread is waiting for signal from predecessor
Squash	Thread is squashed due to dependence violation
Buffer Overflow	Thread made to wait due to lack space to buffer results
Others	Thread stalled due to instruction cache miss, branch mis-prediction, etc.

times more functional units and cache capacity. From these results, we can see that the performance of SMT-4 configuration is quite close to that of the CMP configuration even with much limited resources.

Fig. 6 shows that the SMT-LSQ is stalled for a significant amount of time in some benchmarks due to buffer overflow, thus making it slower than SMT-4 configuration. But in some loops, SMT-LSQ can perform better than SMT-4 configuration. This is because the SMT-LSQ has fewer squashes due to its fine grained dependence checking mechanism. This effect is observed in the benchmark *vpr*. Also, some loops have large number of squashes, in this case it is more beneficial to suspend the threads than to let them execute and later squash. This is because the squashed threads waste resources which could have been allocated to the non-speculative thread. In case of SMT-LSQ large threads are suspended and hence do not waste resources. This effect is observed in some of the loops in the benchmark *twolf*. The performance of SMT-4 can be improved if we have runtime feedback information, so that we can selectively turn off speculative threads on loops with frequent squashes.

In the SMT-LSQ configuration, we used an aggressive 128-entry per thread LSQ which might be unrealistic to implement in reality. Even with this configuration, it is still not able to support some of the loops without overflowing the queue. Fig. 6 shows that the SMT-4 scheme is able to achieve about 3% speedup over such SMT-LSQ configuration. But for loops with an average thread size of more than 150 dynamic instructions, the SMT-4 configuration performs about 14% better than SMT-LSQ configuration.

5 Conclusion

In this paper, we proposed a cache-based scheme to support speculative multithreading in SMT processors. Our two-thread scheme requires 9 bits to be added to each cache line and with this simple modification we can achieve about 10% speedup over the superscalar processors. Our four-thread scheme with slightly more complex hardware can perform about 15% better than superscalar processors. Also, we showed that this cache-based approach can outperform the LSQ based approach by 14% for large loops. From our paper it is clear that speculative threads can be easily supported in SMT processors with minimal changes in hardware.

Acknowledgements. This work was supported in part by the National Science Foundation under grants CCR-015571, CCR-0105574 and EIA-0220021, and grants from Intel.

References

[1] Open research compiler for itanium processor family. http://ipf-orc.sourceforge.net/.

[2] H. Akkary and M. Driscoll. A dynamic multithreading processor. In *MICRO-31*, December 1998.

[3] C. B. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry. Tolerating dependences between large speculative threads via sub-threads. In *Proceedings of the 33th ISCA*, Boston,MA.

[4] M. Franklin and G. S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5), May 1996.

[5] Venkata Krishnan and Josep Torrellas. A chip multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers, Special Issue on Multithreaded Architecture*, September 1999.

[6] C-K Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. ACM SIGPLAN 05 Conference on Programming Language Design and Implementation*, 2005.

[7] P. Marcuello and A. Gonzalez. Exploiting speculative thread-level parallelism on a smt processor. In *Proceedings of the 7th International Conference on High-Performance Computing and Networking*, April 1999.

[8] Jose F. Martnez, Jose Renau, Michael C. Huang, Milos Prvulovic, and Josep Torrellas. Cherry: checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of Micro-35*, Istanbul, Turkey, 2002.

[9] I. Park, B. Falsafi, and T.N. Vijaykumar. Implicitly-multithreaded processors. In *Proceedings of the 30th ISCA*, June 2003.

[10] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th ISCA*, June 2000.

[11] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. The stampede approach to thread-level speculation. In *ACM Transactions on Computer Systems (TOCS)*, volume 23, August 2005.

[12] Dean Tullsen, Susan Eggers, Joel Emer, Henry Levy, Jack Lo, and Rebecca Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd ISCA*, May 1996.

[13] T.N. Vijaykumar, S. Gopal, J.E. Smith, and G. Sohi. Speculative versioning cache. In *IEEE Transactions on Parallel and Distributed Systems*, volume 12, pages 1305–1317, December 2001.

[14] S. Wang, X. Dai, K.S. Yellajyosula, A. Zhai, and P-C Yew. Loop selection for thread-level speculation. In *18th International Workshop on Languages and Compilers for Parallel Computing*, October 2005.

[15] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *Proceedings of the 10th ASPLOS*, Oct 2002.