Using Incorrect Speculation to Prefetch Data in a Concurrent Multithreaded Processor

Ying Chen, Resit Sendag, and David J. Lilja
Department of Electrical and Computer Engineering
Minnesota Supercomputing Institute
University of Minnesota
200 Union St. S.E., Minneapolis, MN 55455, USA
{wildfire, rsgt, lilja}@ece.umn.edu

Abstract

Concurrent multithreaded architectures exploit both instruction-level and thread-level parallelism through a combination of branch prediction and thread-level control speculation. The resulting speculative issuing of load instructions in these architectures can significantly impact the performance of the memory hierarchy as the system exploits higher degrees of parallelism. In this study, we investigate the effects of executing the mispredicted load instructions on the cache performance of a scalable multithreaded architecture. We show that the execution of loads from the wrongly-predicted branch path within a thread, or from a wrongly-forked thread, can result in an indirect prefetching effect for later correctly-executed paths. By continuing to execute the mispredicted load instructions even after the instruction- or thread-level control speculation is known to be incorrect, the cache misses for the correctly predicted paths and threads can be reduced, typically by 42-73%. We introduce the small, fully-associative Wrong Execution Cache (WEC) to eliminate the potential pollution that can be caused by the execution of the mispredicted load instructions. Our simulation results show that the WEC can improve the performance of a concurrent multithreaded architecture up to 18.5% on the benchmark programs tested, with an average improvement of 9.7%, due to the reductions in the number of cache misses.

1. Introduction

A concurrent multithreaded architecture [1] consists of a number of thread processing elements (superscalar cores) interconnected with some tightly-integrated communication network [2]. Each superscalar processor core can use branch prediction to speculatively execute instructions beyond basic block-ending conditional branches. If a branch prediction ultimately turns out to be incorrect, the processor state must be restored to the state prior to the predicted branch and execution is restarted down the correct path. Simultaneously, a concurrent multithreaded architecture can aggressively fork speculative successor threads to further increase the amount of parallelism that can be exploited in an application program. If a speculated control dependence turns out to be incorrect, the non-speculative head thread must kill all of its speculative successor threads.

With both instruction- and thread-level control speculation, a multithreaded architecture may issue many memory references which turn out to be unnecessary since they are issued from what subsequently is determined to be a mispredicted branch path or a mispredicted thread. However, these incorrectly issued memory references may produce an indirect prefetching effect by bringing data or instruction lines into the cache that are needed later by correctly-executed threads and branch paths.

Existing superscalar processors with deep pipelines and wide issue units do allow memory references to be issued speculatively down wrongly-predicted branch paths. However, we go one step further and examine the effects of continuing to execute the loads issued from both mispredicted branch paths and mispredicted threads *even after* the speculative operation is known to be incorrect. We propose the *Wrong Execution Cache* (WEC) to eliminate the potential cache pollution caused by executing the wrong-path and wrong-thread loads. This work shows that the execution of wrong-path or wrong-thread loads can produce a significant performance improvement with very low overhead.

In the remainder of the paper, Section 2 presents an overview of the superthreaded architecture [2], which is the base architecture used for this study. Section 3 describes wrong execution loads and the implementation of the WEC in the base processor. Our experimental methodology is presented in Section 4 with the corresponding results given in Section 5. Section 6 discusses some related work and Section 7 concludes.

2. The Superthreaded Architecture

2.1. Base Architecture Model

The superthreaded architecture (STA) [2] consists of multiple thread processing units (TUs) with each TU connected to its successor by a unidirectional communication ring. Each TU has its own private level-one (L1) instruction cache, L1 data cache, program counter, register file, and execution units. The TUs share the unified second-level (L2) cache. There also is a shared register file that maintains some global control and lock registers. A private memory buffer is used in each thread unit to cache speculative stores for run-time data dependence checking. When multiple threads are executing on an STA processor, the oldest thread in the sequential order is called the *head thread* and all other threads derived from it are called *successor threads*. The program execution starts from its entry thread while all other TUs are idle. When a parallel code region is encountered, this thread activates its downstream thread by forking. This forking continues until there are no idle TUs. When all TUs are busy, the youngest thread delays forking another thread until the head thread retires and its corresponding TU becomes idle. A thread can be forked either speculatively or non-speculatively. A speculatively forked thread will be aborted by its predecessor thread if the speculative control dependence subsequently turns out to be false.

2.2. Thread Pipelining Execution Model

The execution model for the STA architecture is thread pipelining, which allows threads with data and control dependences to be executed in parallel. Instead of speculating on data dependences, the thread execution model facilitates run-time data dependence checking for load instructions. This approach avoids the squashing of threads caused by data dependence violations. It also reduces the hardware complexity of the logic needed to detect memory dependence violations compared to some other CMA execution models [3,4]. As shown in Figure 1 the execution of a thread is partitioned into the *continuation* stage, the *target-store address-generation (TSAG)* stage, the *computation* stage, and the *write-back* stage.

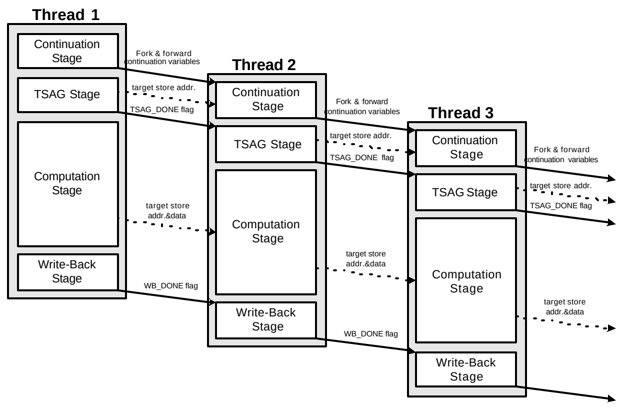


Figure 1. Thread pipelining execution model

The *continuation stage* computes recurrence variables (e.g. loop index variables) needed to fork a new thread on the next thread unit. This stage ends with a *fork* instruction, which initiates a new speculative or non-speculative thread on the next TU. An *abort* instruction is used to kill the successor threads when it is determined that a speculative execution was incorrect. Note that the continuation stages of two adjacent threads can never overlap.

The *TSAG stage* computes the addresses of store instructions on which later concurrent threads *may* have data dependences. These special store instructions are called *target stores* and are identified using conventional data dependence analysis. The computed addresses are stored in the memory buffer of each TU and are forwarded to the memory buffers of all succeeding concurrent threads units.

The *computation stage* performs the actual computation of the loop iteration. If a cross-iteration dependence is detected by checking addresses in memory buffer [2], but the data has not yet arrived from the upstream thread, the out-of-order superscalar core will execute instructions that are independent of the load operation that is waiting for the upstream data value.

In the *write-back stage* all the store data (including target stores) in the memory buffer will be committed and written to the cache memory. The write-back stages are performed in the original program order to preserve non-speculative memory state and to eliminate output and anti-dependences between threads.

3. The Wrong Execution Cache (WEC)

3.1. Wrong Execution

There are two types of *wrong execution* that can occur in a concurrent multithreaded architecture such as the STA processor. The first type occurs when instructions continue to be issued down the path of what turns out to be an incorrectly-predicted conditional branch instruction within a single thread. We refer to this type of execution as *wrong path execution*. The second type of wrong execution occurs when instructions are executed from a thread that was speculatively forked, but is subsequently aborted. We refer to this type of incorrect execution as *wrong thread execution*. Our interest in this study is to examine the effects on the memory hierarchy of load instructions that are issued from both of these types of wrong executions.

3.1.1. Wrong Path Execution

Before a branch is resolved, some load instructions on wrongly-predicted branches may not be ready to be issued because they are waiting either for the effective address to be calculated or for an available memory port. In wrong path execution, however, they are allowed to access the memory system as soon as they are ready even though they are known to be from the wrong path. These instructions are marked as being from a wrong execution path when they are issued so they can be squashed in the pipeline at the write-back stage. A wrong-path load that is dependent upon another instruction that gets flushed after the branch is resolved also is flushed in the same cycle. No wrong-execution store instructions are allowed to alter the memory system since they are known to be invalid.

An example showing the difference between traditional speculative execution and our definition of wrong-path execution is given in Figure 2. There are five loads (A, B, C, D, and E) fetched down the predicted execution path. In a typical pipelined processor, loads A and B become ready and are issued to the memory system speculatively before the branch is resolved. After the branch result is known to be wrong, however, the other three loads, C, D and E, are squashed before being able to access the memory system.

In a system with wrong-path execution, however, ready loads are allowed to continue execution (loads C and D in Figure 2) in addition to the speculatively executed loads (A and B). These *wrong-path loads* are marked as being from the wrong path and are squashed later in the pipeline to prevent them from altering the destination register. However, they are allowed to access the memory to move the value read into the upper levels of the memory hierarchy. Since load E is not ready to execute by the time the branch is resolved, it is squashed as soon as the branch result is known.

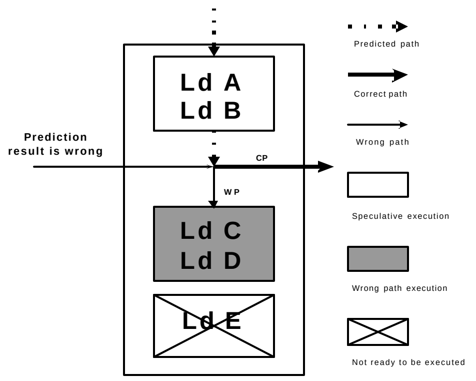
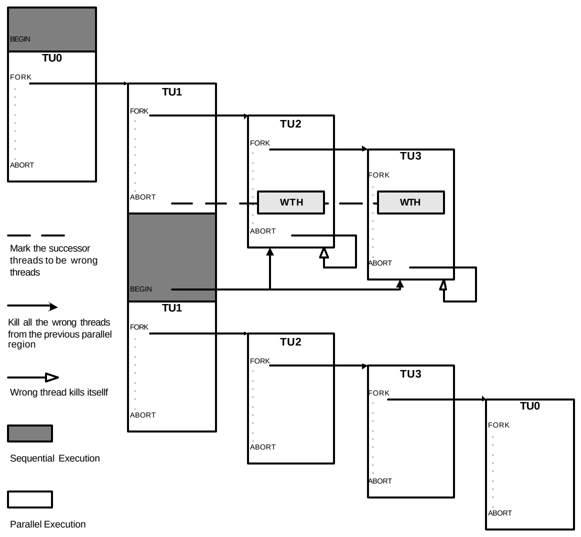


Figure 2. The difference between speculative and wrong-path execution

3.1.2. Wrong Thread Execution

When executing a loop in the normal execution mode of the superthreaded execution model described in Section 2, the head thread executes an *abort* instruction to kill all of its successor threads when it determines that the iteration it is executing satisfies the loop exit condition. To support wrong thread execution in this study, however, the successor threads are marked as wrong threads instead of killing them when the head thread executes an *abort*. These specially-marked threads are not allowed to fork new threads, yet they are allowed to continue execution. As a result, after this parallel region completes its normal execution, the wrong threads continue execution in parallel with the following sequential code. Later, when the wrong threads attempt to execute their own *abort* instructions, they kill themselves before entering the write-back stage.



BEGIN
TU0
FORK
ABORT
TU1
TU2
TU3
WTH
BEGIN
TU0
Mark the successor threads to be wrong threads
Kill all the wrong threads from the previous parallel region
Wrong thread kills itself
Sequential Execution
Parallel Execution

Figure 3. The wrong thread execution model with four TUs

If the sequential region between two parallel regions is not long enough for the wrong threads to determine that they are to be aborted before the beginning of the next parallel region, the *begin* instruction that initiates the next parallel region will abort all of the still-executing wrong threads from the previous parallel region. This modification of the *begin* instruction allows the head thread to fork without stalling. Since each thread's store data are put in a speculative memory buffer local to each TU, and wrong threads do not execute their write-back stages, no stores from the wrong threads can alter the shared memory.

Figure 3 shows this wrong thread execution model with four TUs. Note that although wrong-path and wrong-thread execution have similarities, the main difference between them is that, once a branch is resolved, the ready loads that are not yet ready to execute on a wrong path are squashed, while wrong-thread loads are allowed to continue their execution.

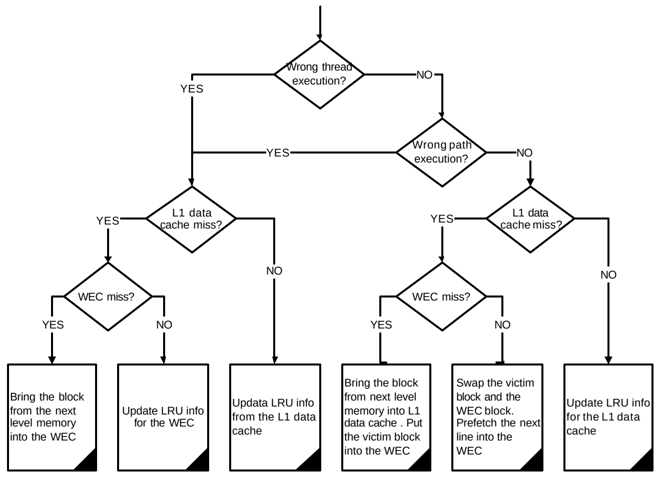
3.2. Operation of the WEC

The indirect prefetching effect provided by the execution of loads down the wrong-paths and the wrong-threads may be able to reduce the number of subsequent correct-path misses. However, these additional wrongly-executed loads may reduce the performance since the cache pollution caused by these loads might offset the benefits of their indirect prefetching effect. This cache pollution can occur when the wrong-execution loads move blocks into the data cache that are never needed by the correct execution path. It also is possible for the cache blocks fetched by the wrong-execution loads to evict blocks that still are required by the correct path. This effect is likely to be more pronounced for low-associativity caches. In order to eliminate this cache pollution, we introduce the *Wrong-Execution Cache* (WEC).

3.2.1. Basic operation of the WEC

The WEC is used to store cache blocks fetched by wrong-execution loads separately from those fetched by loads known to be issued from the correct path, which are stored in the regular L1 data cache. The WEC is accessed in parallel with the L1 data cache. Only those loads that are known to be issued from the wrong-execution, that is, after the control speculation result is known, are handled by the WEC. The data blocks fetched by loads issued before the control speculation is cleared are put into the L1 data cache. After the speculation is resolved, however, a wrong-execution load that causes a miss in both the L1 data cache and the WEC will cause an access to be made to the next level memory. The required block is moved into the WEC to eliminate any cache pollution that might be caused by the wrong-execution load. If a load causes a miss in the L1 data cache, but a hit in the WEC, the block is simultaneously transferred to both the processor and the L1 data cache.

A load from the correct path that hits on a block previously fetched by a wrong-execution load also initiates a next-line prefetch. The block fetched by this next-line prefetch is placed into the WEC. When a correct-execution load causes a miss, the data block is moved into the L1 data cache instead of the WEC, as would be done in a standard cache configuration. The WEC also acts as a victim cache [5] by caching the blocks evicted from the L1 cache by cache misses from the correct execution path. In summary, the WEC is a combination of a prefetch buffer for wrong-execution loads and a victim cache for evictions from the L1 data cache. The operation of the WEC is summarized in Figure 4.



Wrong thread execution?
YES
NO
Wrong path execution?
YES
NO
L1 data cache miss?
YES
NO
L1 data cache miss?
YES
NO
WEC miss?
YES
NO
WEC miss?
YES
NO
Bring the block from the next level memory into the WEC
Update LRU info for the WEC
Updata LRU info from the L1 data cache
Bring the block from next level memory into L1 data cache . Put the victim block into the WEC
Swap the victim block and the WEC block. Prefetch the next line into the WEC
Update LRU info for the L1 data cache

Figure 4. Flowchart of a WEC access.

3.2.2. Incorporating the WEC into the superthreaded architecture

Each TU in the STA used in this study has its own private L1 data cache. In addition, a private WEC is placed in parallel with each of the L1 caches. To enforce coherence among the caches during the execution of a parallel section of code, all possible data dependencies in a thread's execution path are conservatively identified. These potentially shared data items are stored in each TU's private speculative memory buffer until the write-back stage is executed. Updates to shared data items made by a thread during the execution of a parallel section of code are passed to downstream threads via a unidirectional communication ring.

During sequential execution, a simple update protocol is used to enforce coherence. When a cache block is updated by the single thread executing the sequential code, all the other idle threads that cache a copy of the same block in their L1 caches or WECs are updated simultaneously using a shared bus. This coherence enforcement during sequential execution creates additional traffic on the shared bus. This traffic is directed only to what would otherwise be idle caches, however, and does not introduce any additional delays.

4. Experimental Methodology

This study uses the SIMCA (SImulator for Multithreaded Computer Architecture) simulator [6] to model the performance effects of incorporating the WEC into the STA. This simulator is based on the cycle-accurate SimpleScalar simulator, *sim-outorder*, version 3.0 [7]. SIMCA is execution driven and performs both functional and timing simulation.

4.1. TU parameters

Each TU uses a 4-way associative branch target buffer with 1024-entries and a fully associative speculative memory buffer with 128 entries. The distributed L1 instruction caches are each 32KB and 2-way associative. The default unified L2 cache is 512KB, 4-way associative, with a block size of 128 bytes [8]. The L2 cache latency is 12 cycles. The round-trip memory latency is 200 cycles. The L1 cache parameters are varied as described in Section 5. The L1 data cache latency is 1 cycle.

The time required to initiate a new thread (the fork delay) in the STA includes the time required to copy all of the needed global registers to a newly spawned thread's local register file and the time required to forward the program counter. We use a fork delay of four cycles [2] in this study plus two cycles per value to transfer data between threads after a thread has been forked.

4.2. Benchmark Programs

Four SPEC2000 integer benchmarks (*vpr, gzip, mcf, parser*) and two SPEC2000 floating-point benchmarks (*equake, mesa*) are evaluated in this study. All of these programs are written in C. The compiler techniques shown in Table 1 were used to manually parallelize these programs for execution on the STA. The loops chosen for parallelization were identified with run-time profiling as the most time-consuming loops in each program. Table 2 shows the fraction of each program that we were able to parallelize.

Table 1. Program transformations used in manually transforming the code to the thread-pipelining execution model.

Transformations	164. gzip	175. vpr	197. parser	181. mcf	183. equake	177. mesa
Loop Coalescing						✓
Loop Unrolling	✓				✓	✓
Statement Reordering		✓	✓	✓	✓	

Table 2. The dynamic instruction counts of the benchmark programs used in this study, and the fraction of these instructions that were executed in parallel.

Bench -mark	Suite/ Type	Input Set	Whole Benchmark Instruction (M)	Targeted loops Instruction (M)	Fraction Parallelized
175. vpr	SPEC2000 /INT	SPEC test	1126.5	97.2	8.6%
164. gzip	SPEC2000 /INT	Minne -SPEC large	1550.7	243.6	15.7%
181. mcf	SPEC2000 /INT	Minne -SPEC large	601.6	217.3	36.1%
197. parser	SPEC2000 /INT	Minne -SPEC medium	514.0	88.6	17.2%
183. equake	SPEC2000 /FP	Minne -SPEC large	716.3	152.6	21.3%
177. mesa	SPEC2000 /FP	SPEC test	1832.1	319.0	17.3%

The GCC compiler, along with modified versions of the GAS assembler and the GAD loader from the Simplescalar suite, were used to compile the parallelized programs. The resulting parallelized binary code was then executed on the simulator. Each benchmark was optimized at level O3 and run to completion. To keep simulation times reasonable, the

MinneSPEC [9] reduced input sets were used for several of the benchmarks.

4.3. Processor Configurations

The following STA configurations are simulated to determine the performance impact of executing wrong-path and wrong-thread loads, and the performance improvements attributable to the WEC.

orig: This is the baseline supertheaded architecture described in the previous sections.

vc: This configuration adds a small fully-associative victim cache [5] in parallel with the L1 data cache to the *orig* configuration.

wp: This configuration adds more aggressive speculation to a TU's execution as described in Section 3.1.1. It is a good test of how the execution of the loads down the wrong branch path affects the memory system. The thread-level speculation, however, remains the same as in the *orig* configuration.

wth: This configuration is described in detail in Section 3.1.2. Since each thread's store data is put into the speculative memory buffer during a thread's execution, and wrong threads cannot execute their write-back stages, no wrong thread store data alters the memory system. The speculative load execution within a correct TU (superscalar core) remains the same in this configuration as in the *orig* configuration.

wth-wp: This is a combination of the *wp* and *wth* configurations.

wth-wp-vc: This configuration is the *wth-wp* configuration with the addition of a victim cache. It is used to compare against the performance improvement made possible by caching the wrong-path and wrong-thread loads in the WEC.

wth-wp-wec: This is the *wth-wp* configuration with the addition of a small, fully associative WEC in parallel with each TU's L1 data cache. The details of the WEC are given in Section 3.2.1.

nlp: This configuration implements next-line tagged prefetching [10] with a fully associative prefetch buffer, but without any other form of speculative execution. A prefetch is initiated on a miss and on the first hit to a previously prefetched block. The results of these prefetches are put into the prefetch buffer. Tagged prefetching has previously been shown to be more effective than prefetching only on a miss [11]. We used this configuration to compare against the ability of the WEC to successfully prefetch blocks that will be used by subsequently executed loads issued from a correct execution path.

5. Evaluation of Simulation Results

We first examine the baseline performance of the STA followed by an evaluation of the performance of the WEC when using different numbers of TUs. The effects of wrong execution, both with and without a WEC, on the performance of the STA are subsequently examined. We also study the sensitivity of the WEC to several important memory system parameters and analyze the reduction in the number of L1 data cache misses and the increase in the memory traffic due to the WEC.

The overall execution time is used to determine the percentage change in performance of the different configurations tested relative to the execution time of the baseline configuration. Average speedups are calculated using the execution time weighted average of all of the benchmarks [12]. This weighting gives equal importance to each benchmark program independent of its total execution time.

5.1. Baseline Performance of the Superthreaded Architecture

The system parameters used to test the baseline performance, and to determine the amount of parallelism actually exploited in the benchmark programs [13], are shown in Table 3. The size of the distributed 4-way associative L1 data cache in each TU is scaled from 2k to 32k as the number of TUs is varied to keep the total amount of L1 cache in the system constant at 32K.

Table 3. Simulation parameters used for each TU

# of TUs **Issue rate**	**1** **1**	**1** **16**	**2** **8**	**4** **4**	**8** **2**	**16** **1**
Reorder buffer size	8	128	64	32	16	8
INT ALU	1	16	8	4	2	1
INT MULT	1	8	4	2	1	1
FP ALU	1	16	8	4	2	1
FP MULT	1	8	4	2	1	1
L1 data cache size (K)	2	32	16	8	4	2

The baseline for these initial comparisons is a single-thread, single-issue processor, which does not exploit any parallelism. The single-thread-unit, sixteen-issue processor corresponds to a very wide issue superscalar processor that is capable of exploiting only instruction-level parallelism. In the 16TU STA processor, each thread can issue only a single instruction per cycle. Thus, this configuration exploits only thread-level parallelism. The other configurations exploit a combination of both instruction- and thread-level parallelism. Note that the total amount of parallelism available in all of these configurations is constant at 16 instructions per cycle.

Figure 5 shows the amount of instruction- and thread-level parallelism in the parallelized portions of the benchmarks to thereby compare the performance of the STA processor with a conventional superscalar processor. The single TU configuration at the left of each set of bars is capable of issuing 16 instructions per cycle within the single TU. As you move to the right within a group, there are more TUs, but each can issue a proportionally smaller number of instructions per TU so that the total available parallelism is fixed at 16 instructions per cycle.

In these baseline simulations, *164.gzip* shows high thread-level parallelism with a speedup of 14x for the 16TU X 1-issue configuration. A 1TU X 16-issue configuration gives a speedup less than 4x when executing this program. *175.vpr* appears to have more instruction-level than thread-level parallelism since the speedup of the parallelized portion of this program decreases as the number of TUs increases. For most of the benchmarks, the performance tends to improve as the number of TUs increases. This behavior indicates that there is more thread-level parallelism in the parallelized portions of the benchmark programs than simple instruction-level parallelism.

In the cases where the pure superscalar model achieves the best performance, it is likely that the clock cycle time of the very wide issue superscalar processor would be longer than the combined models or the pure STA model. On average, we see that the thread-level parallelization tends to outperform the pure instruction-level parallelization.

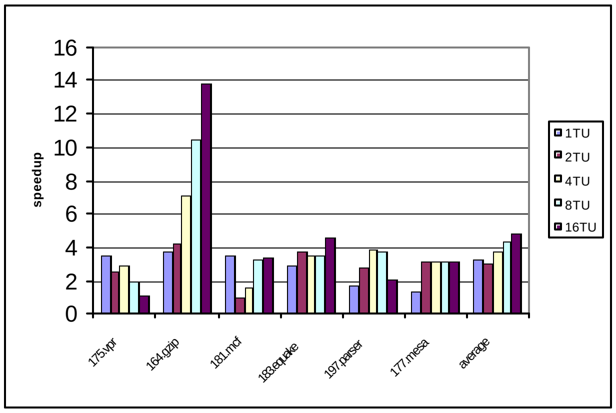


Figure 5. Performance of the STA processor for the parallelized portions of the benchmarks with the hardware configurations shown in Table 3. The baseline configuration is a single-threaded, single-issue processor.

5.2. Performance of the superthreaded processor with the WEC

Based on the results in the previous section, and considering what is expected for future processor development, we use eight TUs, where each TU is an 8-issue out-of-order processor, in the remainder of the study. In some of the experiments, however, we vary the number of TUs as noted to study the impact of varying the available thread-level parallelism on the performance of the WEC.

Each of the TUs has a load/store queue size of 64 entries. The reorder buffer also has 64 entries. The processor has 8 integer ALU units, 4 integer multiply/divide units, 8 floating-point (FP) adders and 4 FP multiply/divide units. The default L1 data cache in each TU is 8KB, direct-mapped, with a block size of 64 bytes. The default WEC has 8 entries and is fully associative with the same block size as the L1 data cache.

Since our focus is on improving the performance of on-chip direct-mapped data caches in a speculative multithreaded architecture, most of the following comparisons for the WEC are made against a victim cache. We also examine the prefetching effect of wrong execution with the WEC by comparing it with next-line tagged prefetching.

5.2.1. The Effect of Varying the Number of TUs

Figure 6 shows the performance of the *wth-wp-wec* configuration as the number of TUs is varied. These results are for the entire benchmark program, not just the parallelized loops. The baseline is the *orig* configuration with a single TU. The speedup of the *wth-wp-wec* configuration can be as much as 39.2% (*183.equake*). For most of the benchmarks, even a two-thread-unit *wth-wp-wec* performs better than the *orig* configuration with 16 TUs.

The single-thread *wth-wp-wec* configuration shows that the WEC can improve the performance significantly, up to 10.4% for *183.equake*, for instance. When more than one TU is used, we see even greater improvements with the WEC due to the larger number of wrong loads issued by executing the wrong threads. For example, in Figure 7, we see that the performance of *181.mcf* improves from 6.2% compared to the baseline configuration when executing with a single TU, to a 20.2% increase over the baseline configuration when using 16 TUs. On average, the performance of the *wth-wp-wec* configuration increases with the number of threads because the total size of the WEC and the L1 cache increases, although the ratio of the WEC size to the L1 data cache size remains constant. Once the total cache and WEC sizes match the benchmark's memory footprint, the performance improvement levels off.

The *175.vpr* program slows down on the *orig* configuration because there is not enough overlap among threads when using more than one TU. As a result, the superthreading overhead overwhelms the benefits of executing the program in parallel. The *181.mcf* program also shows some slowdown for two and four TUs because of contention for TUs.

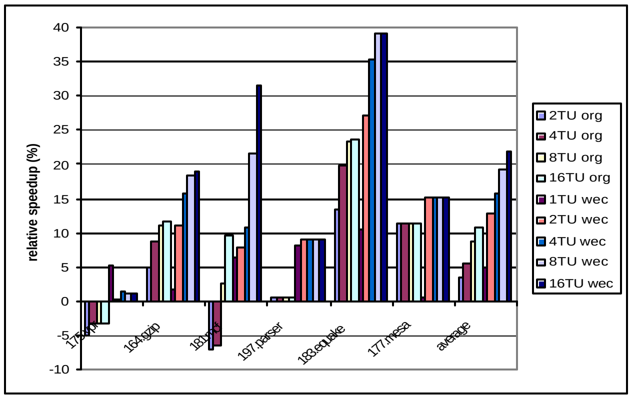


Figure 6. Performance of the *wth-wp-wec* configuration for the entire benchmark programs as the number of TUs is varied. The baseline processor is a STA processor with a single TU.

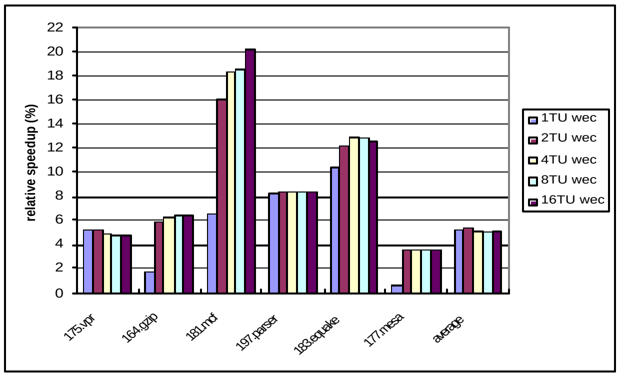


Figure 7. Performance of the *wth-wp-wec* configuration on top of the parallel execution. The baseline processors are one- to 16-TU STA processors with the number of TUs corresponding to the number of threads used in the *wth-wp-wec* configuration.

5.2.2. Performance Improvements Due to the WEC

The previous section showed the performance improvement obtained by executing wrong-path and wrong-thread loads with a WEC in each TU as the total number of available TUs was varied. Figure 8, in contrast, compares the relative speedup obtained by all of the different processor configurations described in Section 4.2 compared to the baseline processor, *orig*. All of these configurations use eight TUs.

This figure shows that the combination of wrong execution plus the WEC (*wth-wp-wec*) gives the greatest speedup of all the configurations tested. The use of only wrong-path or wrong-thread execution alone or in combination (*wp*, *wth*, or *wth-wp*) provides very little performance improvement. When they are used together (*wth-wp*), for instance, the best speedup is only 2.2% (for *183.equake*) while there is some slowdown for *177.mesa*. It appears that the cache pollution caused by executing the wrong loads in these configurations offsets the benefit of their prefetching effect. When the WEC is added, however, the cache pollution is eliminated which produces speedups of up to 18.5% (*181.mcf*), with an average speedup of 9.7%.

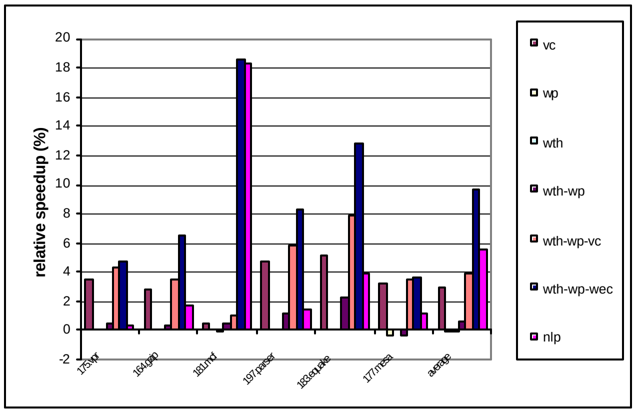


Figure 8. Relative speedups obtained by the different processor configurations with eight TUs. The baseline is the original superthreaded parallel execution with eight TUs.

Compared to a victim cache of the same size, the configurations with the WEC show substantially better performance. While the WEC (*wth-wp-wec*) and the victim cache (*wth-wp-vc*) both reduce conflict misses, the WEC further eliminates the pollution caused by executing loads from the wrong path and the wrong thread.

In addition to this indirect prefetching effect, the WEC also stores the results of the next-line prefetches initiated by a hit to a block in the WEC prefetched through a wrong execution. With both the indirect prefetching and the explicit prefetching, the *wth-wp-wec* performs better than conventional next-line tagged prefetching (*nlp*) with the same size prefetch buffer. Note that the extra hardware cost of both configurations would be approximately the same. On average, conventional next-line prefetching (*nlp*) produces a speedup of 5.5%, while the WEC (*wth-wp-wec*) produces a speedup of 9.7%.

5.3. Parameter Sensitivity Analysis

In this section, we study the effects of varying the L1 data cache associativity, the L1 data cache size, and the WEC size on the performance of the WEC. Each simulation in this section uses eight TUs.

5.3.1. Impact of the L1 Data Cache Associativity

Increasing the L1 cache associativity typically tends to reduce the number of L1 misses for both correct execution [14] and wrong execution [15]. The reduction in misses in the wrong execution paths reduces the number of indirect prefetches issued during wrong execution, which then reduces the performance improvement from the WEC, as shown in Figure 9.

The baseline configuration is the *orig* processor with a direct-mapped and 4-way associative L1 data corresponding to the direct-mapped and 4-way WEC results. When the associativity of the L1 cache is increased, the speedup obtained by the victim cache (*vc*) disappears. However, the configuration with the wrong-execution cache, *wth-wp-wec*, still provides significant speedup. This configuration also substantially outperforms the *wth-wp-vc* configuration, which issues loads from the wrong execution paths, but uses a standard victim cache instead of the WEC.

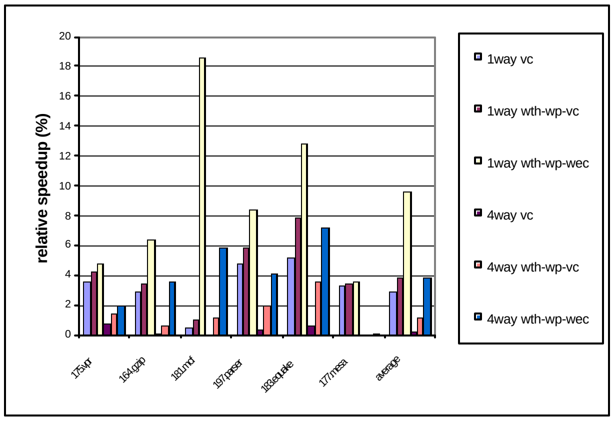


Figure 9. Sensitivity of an eight-TU STA processor with 8-issue superscalar cores and a WEC as the associativity of the L1 data cache is varied (direct-mapped, 4-way).

5.3.2. The Effect of the L1 Data Cache Size

Figure 10 shows the normalized execution times for the *orig* and *wth-wp-wec* configurations when the L1 data cache size is varied. We see that the relative speedup produced by the WEC (*wth-wp-wec*) decreases as the L1 data cache size is increased. However, the WEC size is kept constant throughout this group of simulations so that the relative size of the WEC compared to the L1 data cache is reduced as the L1 size is increased. With a larger L1 cache, the wrong execution loads produce fewer misses compared to the configurations with smaller caches. The smaller number of misses reduces the number of potential prefetches produced by the wrong execution loads, which thereby reduces the performance impact of the WEC.

For all of the test programs, a small 8-entry WEC with an 8K L1 data cache exceeds the performance of the baseline processor (*orig*) when the cache size is doubled, but without the WEC. Furthermore, on average, the WEC with a 4K L1 data cache performs better than the baseline processor with a 32K L1 data cache. These results suggest that incorporating a WEC into the processor is an excellent use of chip area compared to simply increasing the L1 data cache size.

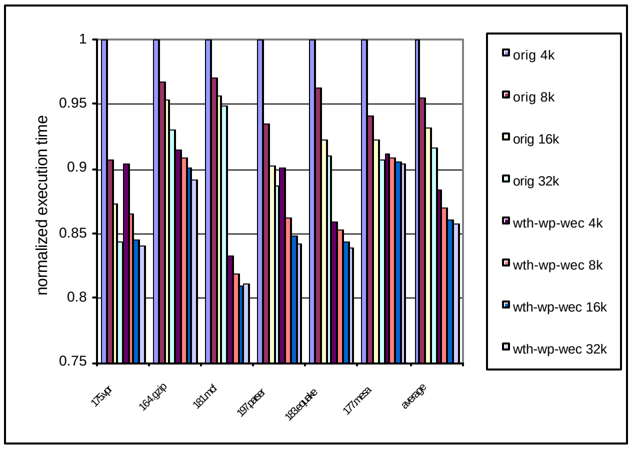


Figure 10. Sensitivity of an eight-TU STA processor with 8-issue superscalar cores and a WEC as the L1 data cache size is varied (4k, 8k, 16k, 32k).

5.3.3. The Effect of the WEC Size

Figure 11 shows that, in general, the configuration that is allowed to issue loads from both the wrong paths and the wrong threads with a 4-entry victim cache (*wth-wp-vc*) outperforms the *orig* configuration with a 16-entry victim cache. Furthermore, replacing the victim cache with a 4-entry WEC causes the *wth-wp-wec* configuration to outperform the configuration with a 16-entry victim cache, *wth-wp-vc*. This trend is particularly significant for *164.gzip*, *181.mcf* and *183.equake*.

Figure 12 compares the WEC approach to a tagged prefetching configuration that uses a prefetch buffer that is the same size as the WEC. It can be seen that the *wth-wp-wec* configuration with an 8-enty WEC performs substantially better than traditional next-line prefetching (*nlp*) with a 32-entry prefetch buffer. This result indicates that the WEC is actually a more efficient prefetching mechanism than a traditional next-line prefetching mechanism.

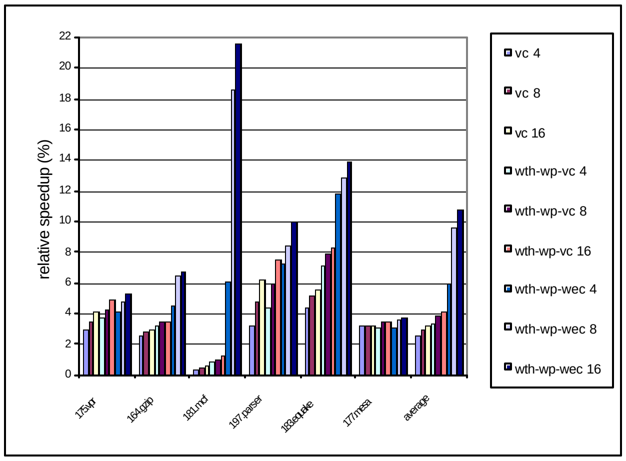


Figure 11. Sensitivity of an eight-TU STA processor with 8-issue superscalar cores and a WEC to changes in the size of the WEC (4, 8, 16 entries) compared to a *vc.*

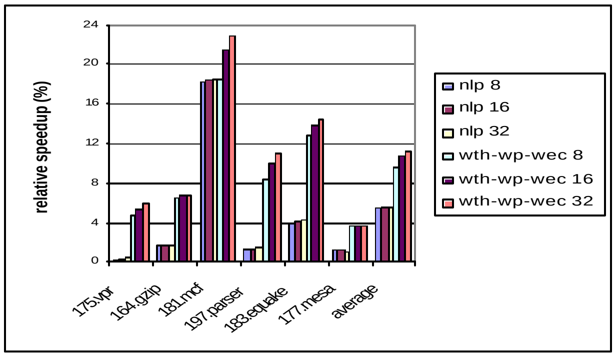


Figure 12. Sensitivity of an eight-TU STA processor with 8-issue superscalar cores and a WEC to changes in the size of the WEC (4, 8, 16 entries) compared to *nlp.*

5.4. L1 Changes in Data Cache Traffic and Misses

Figure 13 shows that the WEC can significantly reduce the number of misses in the L1 data cache. This reduction is as high as 73% for *177.mesa*, although the miss count reduction for *181.mcf* is not as significant as the others. This figure also shows that this reduction in the number of L1 misses comes at the cost of an increase in the traffic between the processor and the L1 cache. This increase in cache traffic is a side effect of issuing more load instructions from both the wrong path and wrong threads. This traffic increase can be as high as 30% in *175.vpr*, with an average increase of 14%. This small average increase in cache traffic would appear to be more than offset by the increase in performance provided by using the WEC, though.

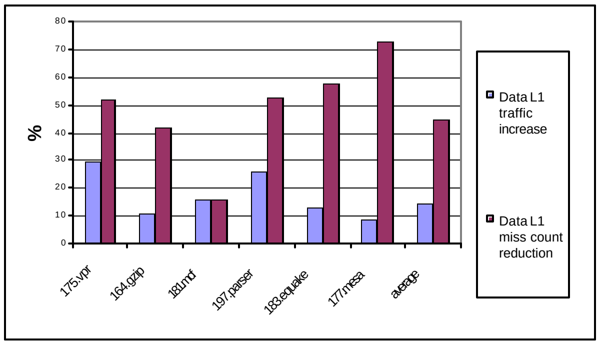


Figure 13. Increases in the L1 cache traffic and the reduction in L1 misses.

6. Related Work

Pierce and Mudge [16] suggested that the additional loads issued from mispredicted branch paths could provide some performance benefits and proposed [17] a wrong-path instruction prefetching scheme in which instructions from both possible branch paths are prefetched. Sendag *et al* [15] examined the impact of wrong-path execution on the data cache in a single-threaded processor. To limit the performance degradation caused by cache pollution, they proposed the *Wrong Path Cache*, which is a combination of a prefetch buffer and a victim cache [5].

While there has been no previous work studying the execution of loads from a mispredicted thread in a multithreaded architecture, a few studies have examined prefetching in the Simultaneous MultiThreading (SMT) architecture. Collins *et al* [18] studied the use of idle thread contexts to perform prefetching based on a simulation of the Itanium processor extended to simultaneous multithreading. Their approach speculatively precomputed future memory accesses using a combination of software, existing Itanium processor features, and additional hardware. Similarly, using idle threads on an Alpha 21464-like SMT processor to pre-execute speculative addresses, and thereby prefetch future values to accelerate the main thread, also has been proposed [19].

These previous studies differ from our work in several important ways. First, this study extends these previous evaluations of single-threaded and SMT architectures to a concurrent multithreading architecture. Second, our mechanism requires only a small amount of extra hardware; no extra software support is needed. Third, while we also use threads that would be idle if there was no wrong thread execution, our goal is not to help the main thread's execution, but rather, to accelerate the future execution of the currently idle threads.

7. Conclusions

In this study, we have examined the effect of executing load instructions issued from a mispredicted branch path (wrong-path) or from a misspeculated thread (wrong-thread) on the performance of a speculative multithreaded architecture. We find that we can reduce the cache misses for subsequent correctly predicted paths and threads by continuing to execute the mispredicted load instructions even after the instruction- or thread-level control speculation is known to be incorrect.

Executing these additional loads causes some cache pollution by fetching never needed blocks and by evicting useful blocks needed for the later correct execution paths and threads. In order to eliminate the potential pollution caused by the mispredicted load instructions, we introduced the small, fully-associative *Wrong Execution Cache* (WEC). Our simulation results show that the WEC can improve the performance of a concurrent multithreaded architecture up to 18.5% on the benchmark programs tested, with an average improvement of 9.7%. This performance improvement comes from reducing the number of cache misses by, typically, 42-73%.

While this study has examined the effects of several parameters on the performance of the WEC, there are still many important factors left to be considered, such as the effects of memory latency, the block size, and the relationship of the branch prediction accuracy to the performance of the WEC.

The WEC proposed in this work is one possible structure for exploiting the potential benefits of executing mispredicted load instructions. Although this current study is based on a multithreaded architecture that exploits loop level parallelism, the ideas presented in this paper can be easily used in all types of multithreaded architectures executing general workloads.

Acknowledgements

This work was supported in part by National Science Foundation grants EIA-9971666 and CCR-9900605, IBM, Compaq's Alpha Development Group, and the Minnesota Supercomputing Institute.

References

[1] T. Ungerer, B. Robic and J. Silc. "Multithreaded Processors," The Computer Journal, Vol.45, No.3, 2002.

[2] J-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P-C. Yew, "The Superthreaded Processor Architecture," IEEE Transactions on Computers, Special Issue on Multithreaded Architectures and Systems, pp. 881-902, September, 1999.

[3] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. "Multiscalar processors," Proceedings of the 22nd Annual International Symposium on Computer Architectures, pages 414-425, June 22-24, 1995

[4] J. G. Steffan and T. C. Mowry. "The potential for thread-level data speculation in tightly-coupled multiprocessors," Technical report, Computer Science Research Institute, University of Toronto, February 1997. Technical Report CSRI-TR-350.

[5] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers," *Proc. 17th Annual International Symposium on Computer Architecture*, Seattle, WA, May 1990, pp. 364-373.

[6] J. Huang, "The SImulator for Multithreaded Computer Architecture," Laboratory for Advanced Research in Computing Technology and Compilers Technical Report No. ARCTiC 00-05, June, 2000.

[7] D. C. Burger, T.M. Austin, and S. Bennett, "Evaluating future Microprocessors: The SimpleScalar Tool Set," *Technical Report CS-TR-96-1308*, University of Wisconsin-Madison, July 1996.

[8] Ying Chen, Resit Sendag, and David J. Lilja, "Using Incorrect Speculation to Prefetch Data in a Concurrent Multithreaded Processor," Laboratory for Advanced Research in Computing Technology and Compilers Technical Report No. ARCTiC 02-09, October, 2002

[9] AJ KleinOsowski, and D. J. Lilja "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *Computer Architecture Letters*, Volume 1, pp. 10-13, May 2002.

[10] J. E. Smith, W.-C. Hsu, "Prefetching in Supercomputer Instruction Caches," In Proceedings of Supercomputing 92, pp. 588-597, 1992.

[11] S. P. VanderWiel and D. J. Lilja, "Data Prefetch Mechanisms," ACM Computing Surveys, Vol. 32, Issue 2, June 2000, pp.174-199.

[12] D. J. Lilja, "Measuring Computer Performance," Cambridge University Press, 2000.

[13] J-Y. Tsai, Z. Jiang, E. Ness and P-C Yew. "Performance Study of a Concurrent Multithreaded Processor," the 4th Inernational Symposium on High Performance Computer Architecture, Feb. 1998.

[14] A. J. Smith, "Cache Memories," *Computing Surveys*, Vol. 14, No. 3, Sept. 1982, pp. 473-530.

[15] R. Sendag, D. J. Lilja, and S. R. Kunkel. "Exploiting the Prefetching Effect Provided by Executing Mispredicted Load Instructions," ACM Euro-Par Conference, August 2002.

[16] J. Pierce and T. Mudge, "The effect of speculative execution on cache performance," IPPS 94, Int. Parallel Processing Symp., Cancun Mexico, pp. 172-179, Apr. 1994

[17] J. Pierce and T. Mudge, "Wrong-Path Instruction Prefetching," Proc. Of 29th Annual IEEE/ACM Symp. Microarchitecture (MICRO-29), Dec. 1996, pp. 165-175

[18] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y-F. Lee, D. Lavery, J. P. Shen. " Speculative Precomputation: Long-range Prefetching of Delinquent Loads," in the 28th Annual International Symposium on Computer Architecture, July 2001.

[19] Luk, H. K. "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," In Proc. of the 28th Annual International Symposium on Computer Architectures, 2001