The Effect of Executing Mispredicted Load Instructions in a Speculative Multithreaded Architecture

Resit Sendag, Ying Chen, 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
{rsgt, wildfire, lilja}@ece.umn.edu

Abstract. Concurrent multithreaded architectures exploit both instruction-level and thread-level parallelism in application programs. A single-threaded sequencing mechanism needs speculative execution beyond conditional branches in order to exploit more instruction-level parallelism. In addition, an aggressive multithreaded architecture should also use thread-level control speculation in order to exploit more thread-level parallelism. The instruction- and thread-level speculative execution of load instructions in a multithreaded architecture system has a greater impact on the performance of the cache hierarchy as the design becomes more aggressive using wider issue processors and more thread units. In this study, we investigate the effects of executing the mispredicted load instructions on the cache performance of a scalable multithreaded computer system. The execution of loads down the wrongly predicted branch path within a thread unit or in a wrongly forked thread can result in an indirect prefetching effect for correct execution. This is possible even after the outcome of a control speculation is known. By allowing mispredicted load instructions to continue execution even after the instruction or thread level control speculation is known to have failed, we show that we can reduce the cache misses for the correctly predicted paths and threads. However, these additional loads also can increase the amount of memory traffic and can pollute the cache. Our results show that the performance of a concurrent multithreaded architecture can be improved as much as 14%, while reducing the number of L1 data cache misses up to 35%.

1 Introduction

A concurrent multithreaded architecture [1] consists of a number of thread processing elements (superscalar cores). In each superscalar processor core, in order to achieve high issue rate, instructions are speculatively executed beyond basic block-ending conditional branches until the branches are resolved. If the prediction was incorrect, the processor state must be restored to the state prior to the predicted branch and execution must be restarted down the correct path. Similarly, to increase the amount of overlap between executing threads, a concurrent multithreaded architecture must aggressively fork speculative successor threads. If any of the speculated control dependences are subsequently found to be false, the thread must kill all the successor threads. With both instruction- and thread-level control speculation, a multithreaded architecture allows many memory references to be issued which turn out to be unnecessary since they are issued from 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 the execution paths for the correct threads. Unfortunately, these additional loads also will increase the memory traffic and may pollute the cache.

Existing superscalar processors with deep pipelines and wide issue units do allow memory references to be issued speculatively down wrongly predicted branch paths. This work, however, proposes to go one step further and examine the effects of continuing to execute the loads down the mispredicted branch path *even after* the branch is resolved. These instructions are marked as being from the mispredicted branch path so that later they can be squashed without altering the target register. In this manner, the processor can continue accessing memory with loads that are known to be from the wrong branch path. At the thread-level, control speculation allows speculative execution of threads. The load instructions issued before the speculation is cleared can affect the cache behavior. In this work, the wrongly predicted threads are allowed to execute instead of being killed. They are marked as wrong threads and are killed when a thread finds itself to be wrongly predicted, or the next parallel region needs to be started. As a result, the wrong threads' execution is overlapped with the execution of the sequential code. This continued execution of incorrectly speculated threads allows more loads to be executed than would execute if the threads were killed at the earliest point at which they are known to be incorrect.

Although this technique issues load instructions very aggressively to produce a significant impact on cache behavior, it has little impact on the implementation of the processor's pipeline and control logic. The execution of wrong-path or wrong-thread loads can make a significant performance improvement with very low overhead when there exists a large disparity between the processor cycle time and the memory speed. However, executing these loads can reduce performance in systems with small data caches and low associativities due to cache pollution.

The remainder of the paper is organized as follows. Section 2 presents an overview of SuperThreaded Architecture [2, 8] (STA), which is the concurrent multithreaded architecture used for this study. Section 3 describes the idea of wrong execution. The experimental methodology is described in Section 4 with the results given in Section 5. Section 6 discusses some related work and Section 7 concludes.

2 The SuperThreaded Architecture (STA)

2.1 Architecture Model

In its general form, the superthreaded architecture (STA) consists of multiple thread processing units. Each unit is connected to its successor by a unidirectional ring, as shown in Figure 1. The thread processing units share the second-level (L2) instruction and data cache. Each unit has a private level-one instruction cache, private or shared level-one data cache, program counter, register file, and execution unit. There also is a shared register file that maintains some global data registers and lock registers. A private memory buffer is used in each thread processing unit to cache speculative stores to support run-time data dependence checking.

When multiple threads are executing on the STA, the oldest thread in the sequential order is called the *head thread*. All other threads derived from it are called *successor threads*. The program execution starts from its entry thread while all other thread units are idle. When a parallel code region is encountered, this thread activates its downstream thread by forking. This forking continues until no thread unit is idle. Then, the youngest thread waits to fork until the head thread retires and the thread unit on which it was executing become idle. As soon as the head thread retires, a new thread is activated. This cycle of idle-active-execute-idle continues until the end of parallel region

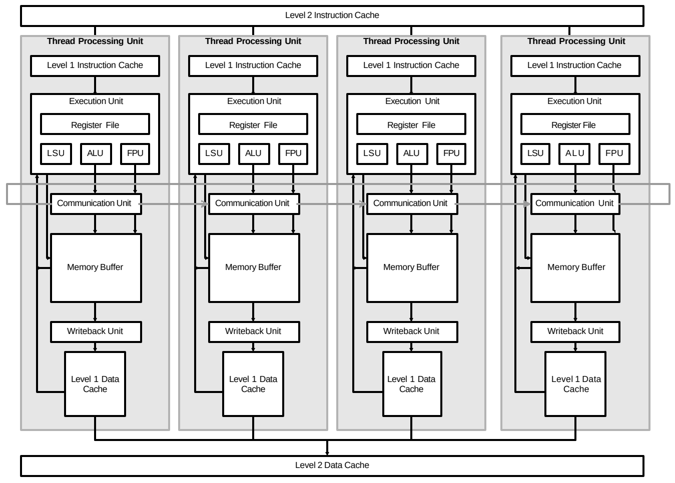

Figure 1. The STA with four threads units.

2.2 Thread Pipelining Execution Model

The execution model for the STA is *thread pipelining*. This execution model allows threads with data and control dependences to be executed in parallel. Instead of speculating on data dependences, it facilitates run-time data dependence checking for load instructions. Thus, it avoids the squashing caused by data dependence violations and reduces the hardware complexity of detecting memory dependence violations compared to some prior CMA models [4], [5]. As shown in Figure 2, 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*.

The main function of the continuation stage is to compute recurrence variables (e.g. loop index variables), and to fork a new thread on the next thread-processing element. This stage ends with a *fork* instruction, which forwards the recurrence variables, as well as the target store addresses and data received from the predecessor threads, to the next thread. A thread can fork a successor thread with or without speculation. If the speculation is ultimately determined to be incorrect, the thread will issue an *abort* instruction to kill the successor threads. The continuation stages of two adjacent threads can never overlap.

The TSAG stage performs address computations for those stores that *may* have data dependences on later concurrent threads. These stores, which are called *target stores*, are identified through a conventional data dependence analysis. The computed addresses are stored in a special memory buffer in each thread unit and forwarded to the memory buffers of all succeeding concurrent thread units.

The computation stage contains the computation from the body of the loop. Reads of data that have cross-iteration dependences are checked against the addresses and values in the memory buffer. If valid data has not yet arrived from an upstream thread, the out-of-order STA core will execute instructions that are independent of the load that is waiting for the upstream data.

In the write-back stage, all the data in the memory buffer, including the values of the target stores, will be committed and written to the cache. The write-back stages are performed in the original program order to preserve the non-speculative memory state and to eliminate output and anti-dependences between threads. After performing the write-back stage, a thread-processing unit can retire the thread. It then becomes idle until it is again scheduled with a new thread.

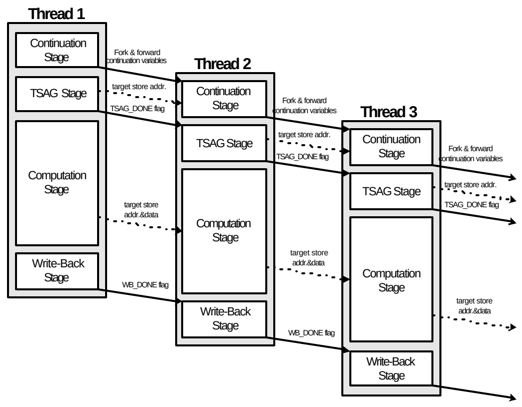

Figure 2. Thread pipelining execution model.

2.3 Compilation process for superthreaded code

All the benchmarks in this study were parallelized manually. First, the *gprof* tool from the GNU suite of Unix tools was used to gather a function-level execution profile. Since these profiles do not provide loop-level details, *tcov*, also from the GNU suite of Unix tools, was used to gather profiles for basic blocks. After choosing the most time-consuming loops in each program, the special superthreaded instructions are inserted by hand to parallelize those loops. The GCC compiler from the Simplescalar suite of tools was used to compile the superthreaded programs. The compiler was modified to transform function calls to *jump-to-label* instructions. A special-purpose parser then was used to replace the function calls with the superthreaded instructions. Finally, the superthreaded assembly code is fed into the Simplescalar GAS assembler and GAD loader to produce the executable binary code. Figure 3 shows the complete compilation process.

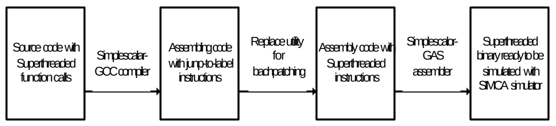
Source code with Superthreaded function calls
Simplescalar-GCC compiler
Assembling code with jump-to-label instructions
Replace utility for bachpatching
Assembly code with Superthreaded instructions
Simplescalar-GAS assembler
Superthreaded binary ready to be simulated with SIMCA simulator

Figure 3. The compilation process for producing superthreaded code.

3 Wrong Execution on the STA

There are two types of *wrong execution* that can occur in a concurrent multithreaded architecture such as the superthreaded architecture. 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. Wrong Path Execution

Loads on wrongly predicted branches that are not ready to be issued before the branch is resolved, either because they are waiting for the effective address calculation or for an available memory port, are issued to the memory system if they become ready after the branch is resolved, even though they are known to be from the wrong path. Instead of being squashed after the branch is resolved, they are allowed to access the memory. These instructions are marked as being from the mispredicted branch path when they are issued so they can be squashed in the write-back stage of the processor pipeline to prevent them from altering the destination register after they access the memory system. Note that 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. In this manner, the processor is allowed to continue accessing memory with loads that are known to be from the wrong branch path. No store instructions are allowed to alter the memory system, however, since they are known to be invalid. The stores that are known to be down the wrong path after the branch is resolved are not executed thereby eliminating the need for an additional speculative write buffer.

An example showing the difference between speculative and wrong-path execution is given in Figure 4. In this example, there are five loads (A, B, C, D, and E) fetched down the predicted path. 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, 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, loads, which become ready, are allowed to continue execution (loads C and D in Figure 4) in addition to the speculatively executed loads (loads A and B). Since they are marked as being from the wrong-path, loads C and D are squashed later in the pipeline so that they will not alter the destination register. 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.

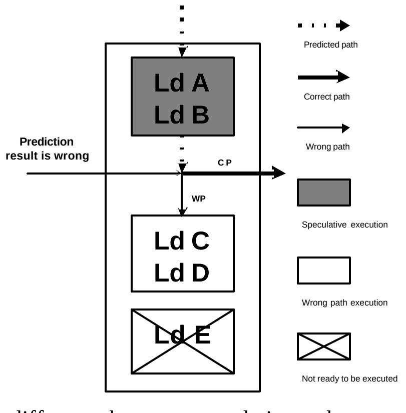

Figure 4. The difference between speculative and wrong-path execution.

3.2 Wrong Thread Execution

The superthreaded execution model must be changed to support wrong thread execution. As described in Section 2, when a thread determines that the iteration it is executing satisfies the loop exit condition, it executes an *abort* instruction to kill all the successor threads. In this study, however, instead of killing them, the successor threads are marked as *wrong threads* 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 finishes, 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 their write-back stages.

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 so that the head thread is able to fork without stalling. Since each thread's store data is put in a speculative memory buffer, and since wrong threads cannot execute their write-back stages, no wrong thread store data alters any data in the cache. The loads from wrong-thread execution will bring data from lower levels of the memory as in wrong-path execution and may have an indirect prefetching effect for later correct threads. However, these wrong-thread loads can pollute the cache as well. Figure 5 shows the wrong thread execution model with four thread units. Although wrong-path and wrong-thread execution has similarities, the main difference between them is that, once a branch is resolved, the loads that are not yet ready to execute on a wrong path are squashed, while wrong-thread loads are allowed to continue their execution.

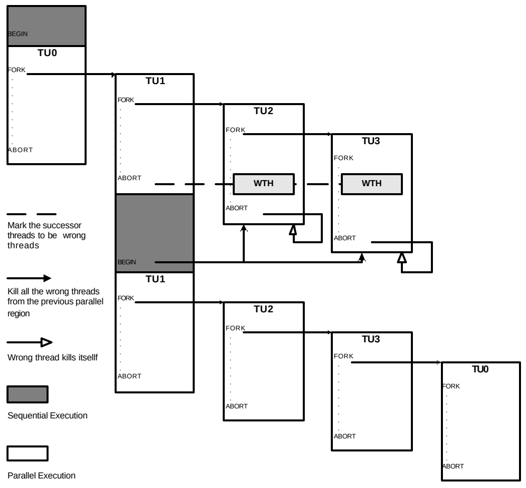
BEGIN
TU0
FORK
ABORT
TU1
TU2
TU3
WTH
WTH
BEGIN
TU1
TU2
TU3
TU0
Mark the successor threads to be wrong threads
Kill all the wrong threads from the previous parallel region
Wrong thread kills itsellf
Sequential Execution
Parallel Execution

Figure 5. Wrong thread execution model with four thread units

4. Experimental Methodology

This study uses SIMCA [7] (SImulator for Multithreaded Computer Architecture) to model the performance effects of wrong-execution on the superthreaded architecture. This simulator is based on version 3.0 of SimpleScalar's *sim-outorder* functional and timing simulator [9].

For all of the simulations in this study, each thread unit uses a 4-way associative branch target buffer of 1024 entries and a fully associative memory buffer of 128 entries. Distributed L1 instruction caches are each 32KB, 2-way associative. The default unified L2 cache is 512KB, 4-way associative with a block size of 128 bytes, unless explicitly mentioned otherwise. The L1 and L2 cache latencies are 1 and 12 cycles, respectively. The round-trip memory latency on a cache miss is 200 cycles.

The time required to initiate a new thread (the fork delay) in the superthreaded architecture 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 and the necessary *target-store* addresses from the memory buffer. 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.

4.1. Benchmark Programs

Four SPEC2000 integer benchmark programs (*gzip, vpr, parser, mcf*) and two SPEC2000 floating point benchmark programs (*equake, mesa*) are evaluated in this study. All of them are written in C.

Some simple complier techniques were used during the parallelization of these benchmark programs, as summarized in Table 1. Each benchmark was compiled with the O3 optimization level and run to completion. To keep the simulation times reasonable, the MinneSPEC [11] reduced input sets were used for several of the benchmark programs, as shown in Table 2. This table also shows the fraction of the code that was parallelized for each program.

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 to Increase Overlap		✓	✓	✓	✓	

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.

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

4.2. Processor Configurations

The following STA configurations were simulated to determine the performance impact of executing wrong-path and wrong-thread loads.

orig: This is the baseline STA explained in the previous sections. Note that the STA allows speculative execution of loads until the control speculation result is known. The results for these loads are speculatively put in the L1 data cache. After the misprediction is known, the processor state is restored to the state prior to the predicted branch. The execution then is restarted down the correct path. In addition, the forking thread immediately kills wrong threads after the speculation is cleared.

wp: This configuration adds more aggressive speculation to a thread unit's execution. The processor allows as many fetched loads as possible to access the memory system regardless of the predicted direction of conditional branches. This configuration is a good test of how the execution of the loads down the wrong branch path affects the memory system. Note that, in contrast to the *orig* configuration, the loads down the mispredicted branch direction are allowed to continue execution even after the branch is resolved. However, they are squashed before being allowed to write to the destination register. Wrong-path stores are not allowed to execute in this configuration to eliminate the need for an additional speculative write buffer. Stores are squashed as soon as the branch result is known. The thread-level speculation, however, remains the same as in configuration *orig*.

wth: This configuration is described in detail in Section 3.2. To summarize, a speculatively forked thread is allowed to continue execution even after it is known to be mispredicted. Wrong threads are squashed before they reach their write-back stage of the thread-execution pipeline to prevent wrongly executed loads from altering the target register after they access the memory system. Since each thread's store data is put in a private local memory buffer during a thread's execution, and wrong threads cannot execute their write-back stages, no wrong thread store data alters the cache. The speculative load execution within a thread unit (i.e., the superscalar core) remains the same as in the *orig* configuration.

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

5. Simulation Results

The simulation results are presented as follows. First, the baseline performance of the STA is given. Next, the effects of wrong-thread and wrong-path execution on the performance of the superthreaded architecture are examined. Finally, several important memory system parameters are varied to determine the sensitivity of the system to these parameters.

The overall execution time is used for the performance metric in this study. Average speedups are calculated by using the execution time weighted average of all of the benchmarks [10].

5.1 Baseline Performance of the STA

The system parameters used to determine the amount of parallelism exploited in the benchmark programs used in this study, and to determine the baseline performance of the STA, are shown in Table 3. The size of the distributed 4-way set-associative L1 data cache size is scaled from 2K to 32K as the number of threads is varied to keep the same total amount of L1 cache in the system. The processor of a single thread unit (1TU) corresponds to a superscalar processor that exploits only instruction-level parallelism. The 16TU configuration, on the other hand, corresponds to a superthreaded architecture that can issue only a single instruction per cycle within each thread unit. Thus, this configuration exploits only thread-level parallelism. The baseline for these initial comparisons is a single-thread, single-issue processor that does not exploit any parallelism.

Table 3. Simulation parameters used for each thread unit.

# of TUs	1	1	2	4	8	16
Issue rate	1	16	8	4	2	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

Figure 6 shows the amount of instruction- and thread-level parallelism in the parallelized portions of the benchmarks to thereby compare the superthreaded processor with a conventional processor. In the baseline simulations, *164.gzip* shows high thread-level parallelism with a speedup of 14*x* for the 16TU X 1-issue configuration, while a 1TU X 16-issue processor gives a speedup less than 4*x*. The speedup of the parallelized portions of *175.vpr* becomes worse as the number of threads is increased because *175.vpr* does not have much thread-level parallelism available. In the cases where the pure superscalar model achieves the best performance, the clock cycle time of this configuration is likely to be longer than the combined models or the pure superthreaded model since a 16-issue superscalar model must use a large instruction reorder buffer for dynamic instruction scheduling.

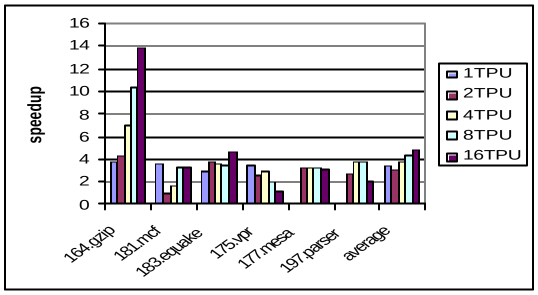

Figure 6. Performance for the transformed portion of superthreaded processors with the hardware configurations shown in Table 3. The baseline configuration is a single-thread, single-issue processor.

5.2. Performance of the Superthreaded Architecture with Wrong-Path and Wrong-Thread Execution

The overall speedups for the benchmarks under test when using wrong-path and wrong-thread execution are given in Figures 7 and 8 for two and eight thread units (TUs), respectively. The speedups are given over the baseline (*orig*) configuration. In Figure 7, for two TUs, the *wth-wp* configuration can result in a 14% performance improvement (for *181.mcf*). This improvement is due to the reduction in the number of correct-path and correct-thread misses that occur by continuing to execute the mispredicted load instructions down the wrong-path and wrong-threads. Figure 8 shows the percent reduction in the number of misses down the correct execution path (i.e., within the correctly predicted thread's execution). We can see that for *181.mcf*, *wth-wp* can reduce the number of misses in the L1 data cache by 35%. The average reduction in the number of L1 data cache misses over all of the benchmark programs is 12.6%.

Figures 9 and 10 show the speedup and miss reduction results for eight TUs, respectively. In Figure 9, we can see some slowdown for some of the benchmarks with the *wp* and *wth* configurations due to the pollution that they cause by bringing never needed blocks into the cache and/or evicting some useful blocks in L1 data cache. This cache pollution offsets the benefits of the indirect prefetching effect by the wrong-path and wrong-thread execution. In Figure 10, the average reduction in L1 data cache misses for eight TUs is 6%. From Figures 7 – 10, it is seen that, of all of the configurations, *wth-wp*, which is a combination of wrong-thread and wrong-path execution, gives the greatest performance improvement.

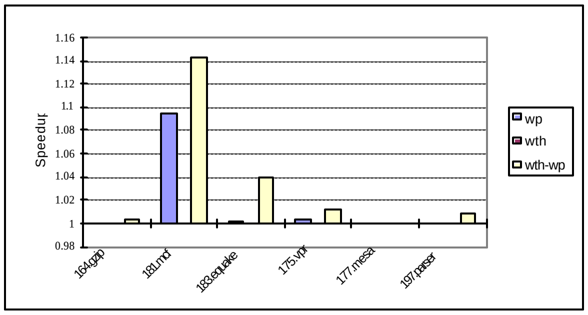

Figure 7. Speedup by wrong-path and wrong-thread execution for two Thread Units (TUs). The baseline is the *orig* configuration with two TUs. The L1 data cache is distributed and 8KB for each TU with 32B blocks and 4-way associativity.

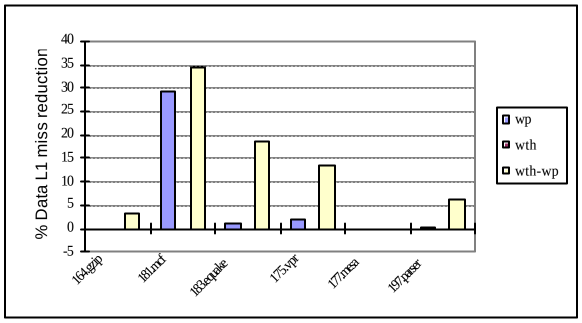

Figure 8. The reduction in L1 data cache misses (percent) due to wrong-path and wrong-thread execution for two thread units.

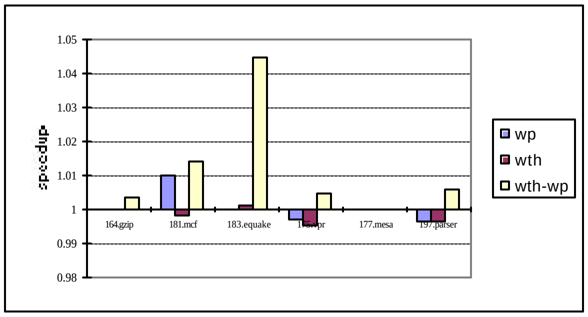

Figure 9. Speedup due to wrong-path and wrong-thread execution for eight thread units (TUs). The baseline is the *orig* configuration with eight TUs. The L1 data cache is distributed and 8KB for each TU with 32B blocks and 4-way associativity.

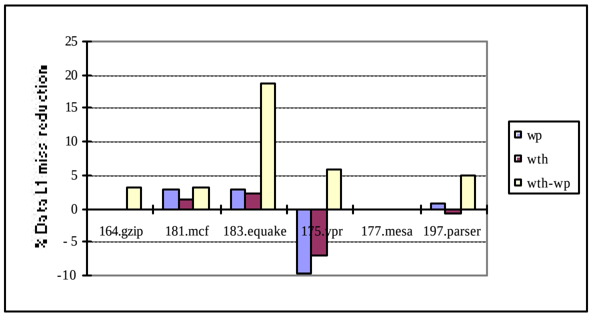

Figure 10. The reduction in L1 data cache misses (percent) due to wrong-path and wrong-thread execution for eight thread units.

5.3 The effect of varying number of thread-processing elements (TUs)

Figure 11 shows the normalized execution times for the *orig*, *wp*, *wth,* and *wth-wp* configurations when the number of TUs is varied. The original 2TU execution is used as the baseline. From these results we see that *177.mesa* does not benefit from thread-level parallelism and wrong-thread and wrong-path execution. *175.vpr* also does not appear to have good thread-level parallelism. The remainder of the benchmarks, however, can take advantage of increasing the number of thread units to reduce the overall execution time. Additionally, the *wth-wp* configuration further reduces the overall execution time for these benchmark programs.

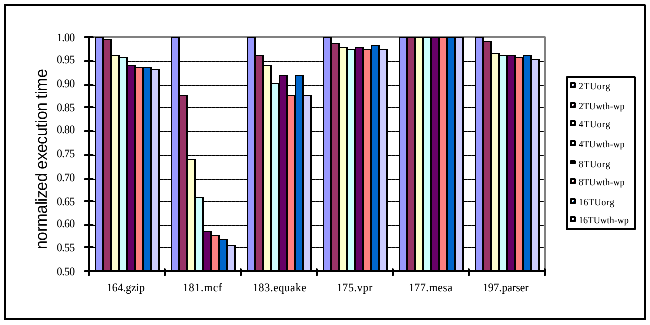

Figure 11. Normalized execution times for various number of thread units (2,4,8,16)

5.4 Parameter Sensitivity Analysis

In this section, we study the effects of varying the L1 cache size and associativity on the performance of the *wth, wp,* and *wth-wp* configurations. Each simulation in this section uses eight thread units.

5.4.1 L1 data cache size

Figure 12 shows the normalized execution times when the L1 data cache size is varied. With a larger L1 data cache, we see that the performance increases because of reductions in the number of correct path cache misses, which is shown in Figure 14. As shown in Figure 13, the indirect prefetching effect provided by the *wth-wp* configuration can increase performance up to 4.6% (*183.equake*). As the size of cache increases, the speedup tends to reduce. This behavior occurs because, when cache size is increased, the number of misses during the wrong execution reduces as well. As a result, there is a smaller prefetching effect which reduces the number of misses on the correct path. On the other hand, the larger cache size causes fewer correct path misses and less cache pollution due to wrong execution, which further decreases the number of misses on the correct path. Figure 15 shows that the two trends have different tradeoffs on different benchmarks.

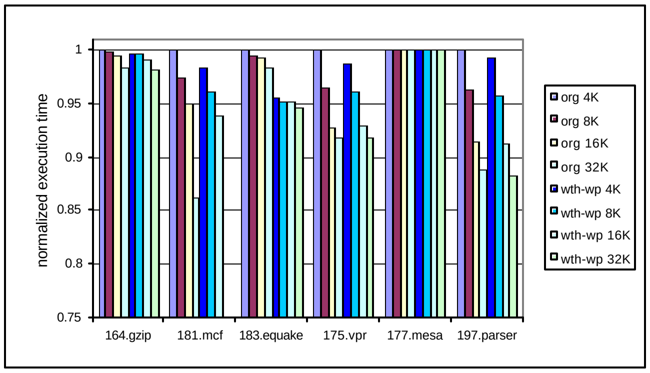

Figure 12. Normalized execution times with different L1 cache sizes (4K, 8K, 16K, 32K).

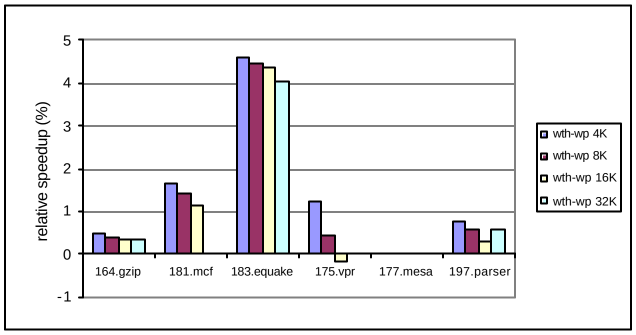

Figure 13. Relative speedups for different L1 data cache sizes (4K, 8K, 16K, 32K). The baseline configuration is chosen to match the corresponding L1 data cache size.

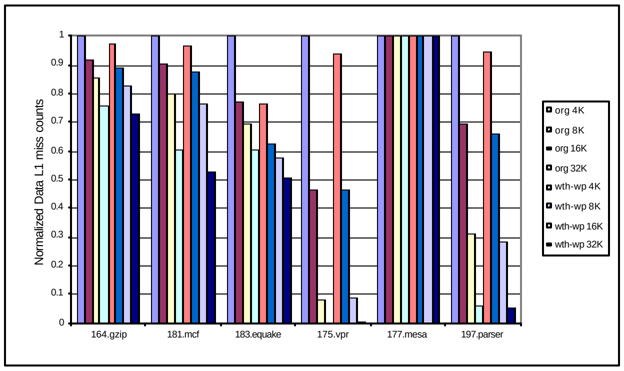

Figure 14. Normalized L1 data cache miss counts on the correct execution path with different cache sizes (4K, 8K, 16K, 32K).

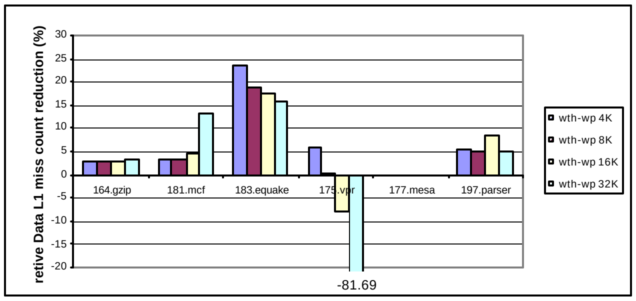

Figure 15. The relative reduction in the number of misses in the data L1 cache for different L1 data cache sizes (4K, 8K, 16K, 32K). The baseline configuration is chosen to match the corresponding L1 data cache size.

5.4.2 L1 data cache associativity

As the associativity of the L1 cache increases, the benchmark performance also tends to increase, as shown in Figure 16. This increase is due to a reduction in the number of misses on the correct path miss, which is shown in Figure 18. Cache pollution due to the wrong execution is significant when the cache associativity is small. Although the number of indirect prefetches due to wrong execution increases with a smaller associativity, the pollution caused by the wrong execution offsets the benefit of the indirect prefetching effect. As we can see from Figure 17, the relative speedup from the wrong execution increases when the cache associativity increases. In this figure, we see speedups up to 4.4% for *183.equake*, for example., The number of misses on both the correct path and the wrong path decreases when the cache associativity increases. As a result, the wrong thread's indirect prefetching effect is more prominent. That is, wrong execution tends to be more effective in reducing correct path misses as cache associativity increases. We conclude that eliminating the pollution caused by the execution of mispredicted load instructions for the low associativity caches is important to increase the benefit from wrong-path and wrong-thread execution.

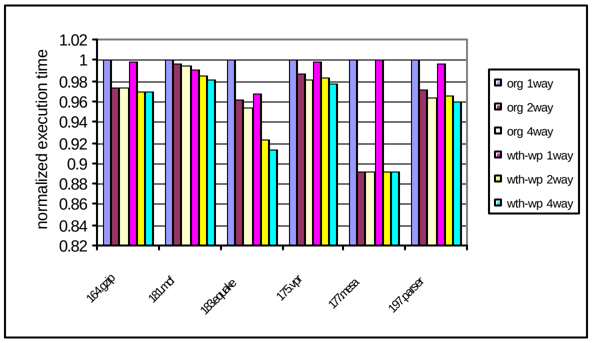

Figure 16. Normalized execution time with different L1 associativities (1way, 2way, 4way)

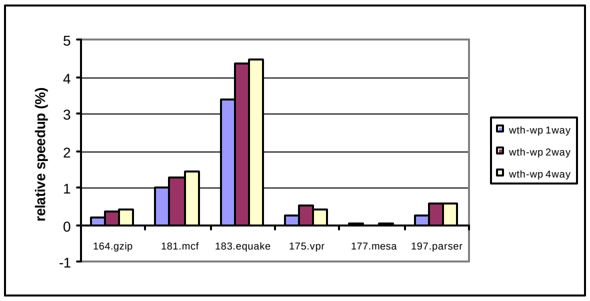

Figure 17. Relative speedup for different L1 data cache associativities (1way, 2way, 4way). The baseline configuration is chosen to match the corresponding L1 data cache associativity.

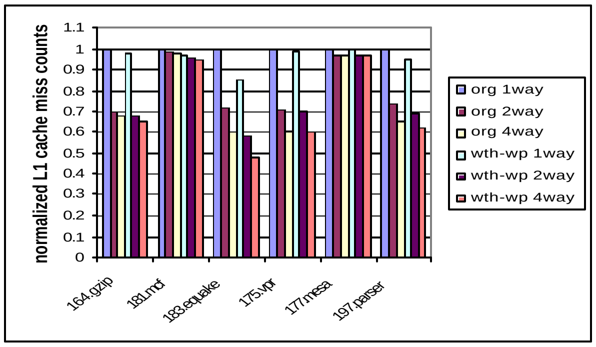

Figure 18. Normalized L1 data cache miss counts for different L1 data cache associativities (1way, 2way, 4way).

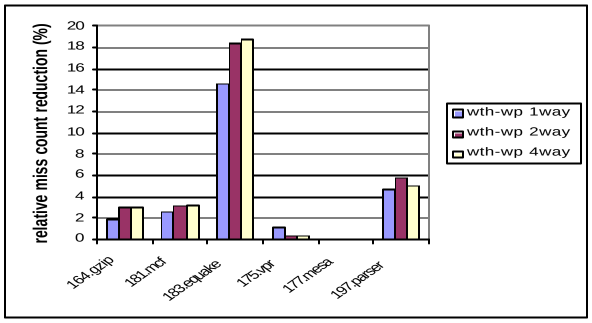

Figure 19. The relative reduction in the number of misses in the data L1 cache for different L1 data cache associativities (1way, 2way, 4way). The baseline configuration is chosen to match the corresponding L1 data cache associativity.

5.5 Memory Traffic

As shown in Figure 20, the number of load references to the L1 data cache, and the traffic between the L1 and L2 caches, increases due to the execution of wrong-path and wrong-thread loads. There is a 16.5% increase in the number of L1 data cache accesses and a 10.0% increase in the number of unified L2 cache accesses on average. The *164.gzip*, *181.mcf,* and *183.equake* programs originally had large L2 cache traffic. As a result, executing the wrong loads in the *wth-wp* configuration does not add much additional traffic. For *175.vpr* and *197.parser*, in contrast, the L2 cache traffic was originally relatively small. Consequently, the additional wrong loads executed in with the *wth-wp* configuration increases the traffic by a relatively large amount.

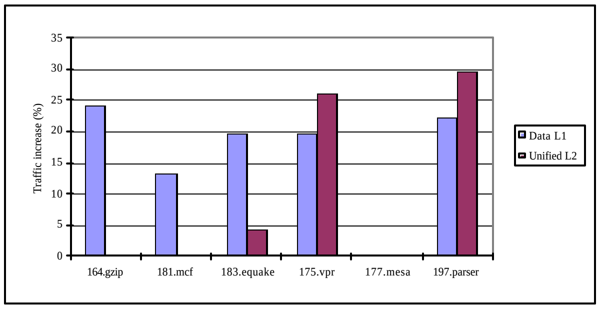

Figure 20. Changes in the relative L1 and L2 cache traffic due to executing wrong loads in the *wth-wp* configuration compared to the *orig* configuration

with an 8K, 4-way associative L1 cache, and a 512K L2 cache when using 8 thread units.

6 Related Work

Many studies have examined the effect of speculative execution on single-thread architectures. Intuitively, speculative execution may significantly increase memory traffic. Pierce and Mudge's [12] study shows that this intuition is not necessarily true for mispredicted loads from deep speculative execution. On the contrary, their study showed that executing these loads may have some potential benefits. This previous work did not quantitatively evaluate the potential benefits, however. Pierce and Mudge also proposed a wrong-path instruction prefetching scheme [13] in which instructions from both possible branch paths are prefetched. Their result showed that wrong-path prefetching can be surprisingly effective in reducing instruction cache misses. Sendag *et al* [3] examined the impact of wrong-path execution on the data cache in a single-threaded processor. This study quantified the benefits and tradeoffs of wrong path execution. Our speculative execution mechanism in this paper is based on a multithreaded architecture, which adds inter-thread speculation (wrong-thread execution) to the intra-thread speculation (wrong-path execution) of a single-threaded processor.

While there has been no previous work that has examined the impact of executing loads from a mispredicted thread in a multithreaded architecture, a few studies have examined prefetching in the Simultaneous MultiThreading (SMT) architecture. Collins *et al* [6] studied the use of idle thread contexts to perform prefetching based on a simulation of the Itanium processor that had been extended to perform simultaneous multithreading. Their approach speculatively precomputed future memory accesses using a combination of software, existing Itanium processor features, and additional hardware support. 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 [14].

These previous studies differ from our work in this paper in several important ways. First, this current 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, which is transparent to the processor; no extra software support is needed.

7 Conclusions

In this paper, we examined the effect of executing mispredicted load instructions from the wrong-path, and from a wrongly forked thread, on the performance of a speculative multithreaded architecture. We find that by continuing to execute the mispredicted load instructions, we can reduce the misses for subsequent correct execution paths and threads. We also find that there is a pollution effect caused by bringing never needed blocks into the cache and by evicting useful blocks needed for the later correct execution. We show that the indirect prefetching effect of wrong execution can improve the performance of a concurrent multithreaded architecture as much as 14%, while reducing the number of misses up to 35%. The low associativity caches, on the other hand, although having more misses, cannot benefit from the wrong execution as much since the cache pollution caused by the wrong execution can offset the prefetching effect. In order to get more benefit from wrong execution for low-associativity caches, we must eliminate the pollution that they cause. This can be done by directing their results into a small buffer as is done in [3]. Examination of the performance with such a mechanism is left to future work. 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.

Acknowledgment

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] Theo Ungerer, Borut Robic ad Jurij Silc. "Multithreaded Processors". The Computer Journal, Vol.45, No.3, 2002

[2] Jenn-Yuan Tsai, Jian Huang, Christoffer Amlo, David J. Lilja, and Pen-Chung Yew, "The Superthreaded Processor Architecture". IEEE Transactions on Computers, Special Issue on Multithreaded Architectures and Systems, September, 1999.

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

[4] G.S. Sohi, S.E. Breach, and T.N. Vijaykumar. "Multiscalar processors" International Symposium on Computer Architecture, pages 414-425, June22-24, 1995

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

[6] 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", International Symposium on Computer Architecture , July 2001.

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

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

[9] 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.

[10] David J. Lilja, "Measuring Computer Performance", Cambridge University Press, 2000

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

[12] 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

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

[14] H. K. Luk, "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors" International Symposium on Computer Architecture, 2001