Exploiting the Prefetching Effect Provided by Executing Mispredicted Load Instructions

Resit Sendag[1], David J. Lilja[1], and Steven R. Kunkel[2]

[1] Department of Electrical and Computer Engineering
Minnesota Supercomputing Institute
University of Minnesota
200 Union St. S.E., Minneapolis, MN 55455, USA
{rsgt, lilja}@ece.umn.edu
[2] IBM, Rochester, MN, USA
srkunkel@us.ibm.com

Abstract. As the degree of instruction-level parallelism in superscalar architectures increases, the gap between processor and memory performance continues to grow requiring more aggressive techniques to increase the performance of the memory system. We propose a new technique, which is based on the wrong-path execution of loads far beyond instruction fetch-limiting conditional branches, to exploit more instruction-level parallelism by reducing the impact of memory delays. We examine the effects of the execution of loads down the wrong branch path on the performance of an aggressive issue processor. We find that, by continuing to execute the loads issued in the mispredicted path, even after the branch is resolved, we can actually reduce the cache misses observed on the correctly executed path. This wrong-path execution of loads can result in a speedup of up to 5% due to an indirect prefetching effect that brings data or instruction blocks into the cache for instructions subsequently issued on the correctly predicted path. However, it also can increase the amount of memory traffic and can pollute the cache. We propose the Wrong Path Cache (WPC) to eliminate the cache pollution caused by the execution of loads down mispredicted branch paths. For the configurations tested, fetching the results of wrong path loads into a fully associative 8-entry WPC can result in a 12% to 39% reduction in L1 data cache misses and in a speedup of up to 37%, with an average speedup of 9%, over the baseline processor.

1 Introduction

Several methods have been proposed to exploit more instruction-level parallelism in superscalar processors and to hide the latency of the main memory accesses, including speculative execution [1-7] and data prefetching [8-21]. To achieve high issue rates, instructions must be fetched beyond the basic block-ending conditional branches. This

B. Monien and R. Feldmann (Eds.): Euro-Par 2002, LNCS 2400, pp. 468–480.

can be done by speculatively executing instructions beyond branches until the branches are resolved. This speculative execution will allow many memory references to be issued that turn out to be unnecessary since they are issued from the mispredicted branch path. 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 instructions that are subsequently issued along correct execution path. On the other hand, these incorrectly issued memory references will increase the amount of memory traffic and can potentially pollute the cache with unneeded cache blocks [2].

Existing processors with deep pipelines and wide issue units do allow memory references to be issued speculatively down wrongly-predicted branch paths. In this study, however, we 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*. That is, we allow all speculatively issued loads to access the memory system if there is an available memory port. 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 target register after they access the memory system. 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.

While this technique very aggressively issues load instructions to produce a significant impact on cache behavior, it has very little impact on the implementation of the processor's pipeline and control logic. The execution of wrong-path 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. This cache pollution occurs when the wrong-path 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-path loads to evict blocks that still are required by the correct path.

In order to eliminate the cache pollution caused by the execution of the wrong-path loads, we propose the *Wrong Path Cache (WPC)*. This small fully-associative cache is accessed in parallel with the L1 cache. It buffers the values fetched by the wrong-path loads plus the blocks evicted from the data cache. Our simulations show that the WPC can be very effective in eliminating the pollution misses caused by the execution of wrong path loads while simultaneously reducing the conflict misses that occur in the L1 data cache.

The remainder of the paper is organized as follows -- Section 2 describes the proposed wrong path cache. In Section 3, we present the details of the simulation environment with the simulation results given in Section 4. Section 5 discusses some related work with the conclusions given in Section 6.

2 Wrong Path Cache (WPC)

For small low-associativity data caches, the execution of loads down the incorrectly-predicted branch path can reduce performance since the cache pollution caused by

these wrong-path loads might offset the benefits of their indirect prefetching effect. To eliminate the pollution caused by the indirect prefetching effect of the wrong-path loads, we propose the Wrong Path Cache (WPC). The idea is simply to use a small fully associative cache that is separate from the data cache to store the values returned by loads that are executed down the incorrectly-predicted branch path. Note that the WPC handles the loads that are *known* to be issued from the wrong path, that is, after the branch result is known. The loads that are executed before the branch is resolved are speculatively put in the L1 data cache.

If a wrong-path load causes a miss in the data cache, the required cache block is brought into the WPC instead of the data cache. The WPC is queried in parallel with the data cache. The block is transferred simultaneously to the processor and the data cache when it is not in the data cache but it is in the WPC. When the address requested by a wrong-path load is in neither the data cache nor the WPC, the next cache level in the memory hierarchy is accessed. The required cache block is then placed into the WPC only to eliminate the pollution in the data cache that could otherwise be caused by the wrong-path loads. Note that misses due to loads on the correct execution path, and misses due to the loads issued from the wrong path before the branch is resolved, move the data into the data cache but not into the WPC.

The WPC also caches copies of blocks recently evicted by cache misses. That is, if the data cache must evict a block to make room for a newly referenced block, the evicted block is transferred to the WPC, as is done in the victim cache [9].

3 Experimental Setup

3.1 Microarchitecture

Our microarchitectural simulator is built on top of the SimpleScalar toolset [22], version 3.0. The simulator is modified to compare the processor configurations described in Section 3.2. The processor/memory model used in this study is an aggressively pipelined processor capable of issuing 8 instructions per cycle with out-of-order execution. It has a 128-entry reorder buffer with a 64-entry load/store buffer. The store forwarding latency is increased to 3 cycles in order to compensate for the added complexity of disambiguating loads and stores in a large execution window. There is a 6-cycle branch misprediction penalty. The processor has 8 integer ALU units, 2-integer MULT/DIV units, 4 load/store units, 6-FP Adders and 2-FP MULT/DIV units. The latencies are: ALU=1 cycle, MULT=3 cycles, integer DIV=12 cycles, FP Adder=2 cycles, FP MULT=4 cycles, and FP DIV=12 cycles. All the functional units, except the divide units, are fully pipelined to allow a new instruction to initiate execution each cycle.

The processor has a first-level 32 KB, 2-way set associative instruction cache. Various sizes of the L1 data cache (4KB, 8KB, 16KB, 32KB) with various associativities (direct-mapped, 2-way, 4-way) are examined in the following simulations. The first-level data cache is non-blocking with 4 ports. Both caches have block sizes of 32 bytes and 1-cycle hit latency. Since the memory footprints of the benchmark programs used in this paper are somewhat small, a relatively small 256K 4-way associative unified L2 cache is used for all of the experiments in order to produce significant L2 cache activity. The L2 cache has 64-byte blocks and a hit

latency of 12 cycles. The round-trip main memory access latency is 200 cycles for all of the experiments, unless otherwise specified. We model the bus latency to main memory with a 10 cycle bus occupancy per request. Results are shown for bus bandwidth of 8 bytes/cycle. The effect on the WPC performance of varying the cache block size is examined in the simulations. There is a 64-entry 4-way set associative instruction TLB and 128-entry 4-way set associative data TLB, each with a 30-cycle miss penalty. For this study, we used the GAp branch predictor [24, 25]. The predictor has a 4K-entry Pattern History Table (PHT) with 2-bit saturating counters.

3.2 Processor Configurations Tested

The following superscalar processor configurations are simulated to determine the performance impact of executing wrong-path loads, and the performance contributions of the Wrong Path Cache. The configurations, *all*, *vc*, and *wpc*, are modifications of the SimpleScalar [22] baseline processor described above.

orig: This configuration is the SimpleScalar baseline processor. It is an 8-issue processor with out-of-order execution and support for speculative execution of instructions issued from a predicted branch path. Note that this processor can execute loads from a mispredicted branch path. These loads can potentially change the contents of the cache, although they cannot change the contents of any registers. These wrong-path loads are allowed to access the cache memory system until the branch result is known. After the branch is resolved, they are immediately squashed and the processor state is restored to the state prior to the predicted branch. The execution then is restarted down the correct path.

all: In this configuration, 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. Wrong-path loads 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 as in the *orig* configuration, they are allowed to access the memory. However, they are squashed before being allowed to write to the destination register. 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. Wrong-path stores are not allowed to execute and are squashed as soon as the branch result is known.

orig_vc: This configuration is the *orig* configuration (the baseline processor) with the addition of an 8-entry victim cache.

all_vc: This configuration is the *all* configuration with the addition of an 8-entry victim cache. It is used to compare against the performance improvement made possible by caching of the wrong-path loads in the WPC.

wpc: This configuration adds an 8-entry Wrong Path Cache (WPC) to the *all* configuration.

3.3 Benchmark Programs

The test suite used in this study consists of the combination of SPEC95 and SPEC2000 benchmark programs. All benchmarks were compiled using gcc 2.6.3 at optimization level O3 and each benchmark ran to completion. The SPEC2000 benchmarks are run with the MinneSPEC input data sets to limit their total simulation time while maintaining the fundamental characteristics of the programs' overall behaviors [23].

4 Results

The simulation results are presented as follows. First, the performances of the different configurations are compared using the speedups relative to the baseline (*orig*) processor. Next, several important memory system parameters are varied to determine the sensitivity of the WPC to these parameters. The impact of executing wrong-path loads both with and without the WPC also is analyzed.

Having used small or reduced input sets to limit the simulation time, most of the results are given for a relatively small L1 data cache to mimic more realistic workloads with higher miss rates. The effect of different cache sizes is investigated in Section 4.2. In this paper, our focus is on improving the performance of on-chip direct-mapped data caches. Therefore, most of the comparisons for the WPC are made against a victim cache [9]. We do investigate the impact of varying the L1 associativity in Section 4.2, however.

4.1 Performance Comparisons

4.1.1 Speedup Due to the WPC

Figure 1 shows the speedups obtained relative to the *orig* configuration when executing each benchmark on the different configurations described in Section 3.2. The WPC and the victim cache each have eight entries in those configurations that include these structures.

Of all of the configurations, *wpc*, which executes loads down the wrong branch path with an 8-entry WPC, gives the greatest speedup. From Figure 1, we can see that, for small caches, the *all* configuration actually produces a slowdown due to the large number of wrong-path loads polluting the L1 cache. However, by adding the WPC, the new configuration, *wpc*, produces the best speedup compared to the other configurations. In particular, *wpc* outperforms the *orig_vc* and *all_vc* configurations, which use a simple victim cache to improve the performance of the baseline processor. While both the WPC and the victim cache reduce the impact of conflict misses in the data cache by storing recent evictions near the processor, the WPC goes further by acting like a prefetch buffer and thus preventing pollution misses due to the indirect prefetches caused by executing the wrong-path loads in the *all* configuration.

While we will study the effect of different cache parameters in later sections, Figure 2 shows the speedup results for an 8KB L1 data cache with 4-way associativity. When increasing the associativity of the L1 cache, the speedup obtained by the *orig_vc* seen in Figure 1 disappears. However, the *wpc* still provides

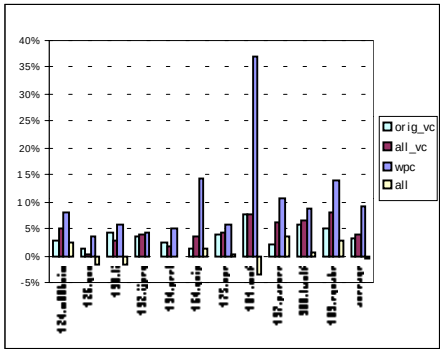

Fig. 1. The Wrong Path Cache (*wpc*) produces consistently higher speedups than the victim cache (*vc*) or the *all* configuration, which does not have a WPC but does execute all ready wrong–path loads if there is a free port to the memory system. The data cache is 8KB direct-mapped and has 32-byte blocks. All speedups are relative to the baseline (*orig*)processor.

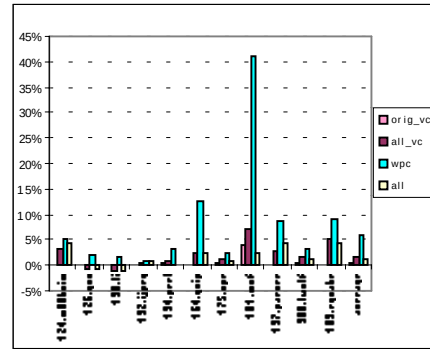

Fig. 2. With a data cache of 8KB with 4-way associativity, the speedup obtained by *orig_vc* disappears. However, *wpc* continues to provide significant speedup and substantially outperforms the *all_vc* configuration. The *all* configuration also shows significant speedup for some benchmarks. The data cache has 32-byte blocks. All speedups are relative to the baseline (*orig*) processor.

significant speedup as the associativity increases and it substantially outperforms the *all_vc* configuration. The *mcf* program shows generally poor cache behavior and increasing the L1 associativity does not reduce its miss rate significantly. Therefore, we see that the speedup produced by the *wpc* for *mcf* remains the same in Figures 1 and 2. As expected, a better cache with lower miss rates reduces the benefit of the *wpc*. From Figure 2, we also see that the *all* configuration can produce some speedup. There is still some slowdown for a few of the benchmarks due to pollution from the wrong path execution of loads. However, the slowdown for the *all* configuration is less than in Figure 1, where the cache is direct-mapped.

4.1.2 A Closer look at the WPC Speedups

The speedup results shown in Figures 1 and 2 can be explained at least partially by examining what levels of the memory hierarchy service the memory accesses. Figure 3 shows that the great majority of all memory accesses in the benchmark programs are serviced by the L1 cache, as is to be expected. While a relatively small fraction of the memory accesses cause misses, these misses add a disproportionately large amount of time to the memory access time. The values for memory accesses that miss in the L1 cache must be obtained from one of three possible sources, the wrong-path cache (WPC), the L2 cache, or the memory. Figure 3 shows that a substantial fraction of the misses in these benchmark programs are serviced by the WPC. For example, 4% of all memory accesses issued by *twolf* are serviced by the WPC. However, this fraction corresponds to 32% of the L1 misses generated by this program. Similarly, 3.3% of *mcf*'s memory accesses, and 1.9% of *equake*'s, are serviced by the WPC, which corresponds to 21% and 29% of their L1 misses, respectively. Since the WPC is accessed in parallel with the L1 cache, misses serviced by the WPC are serviced in the same amount of time as a hit in the L1 cache, while accesses serviced by the L2 cache require 12 cycles and accesses that must go all the way to memory require 200

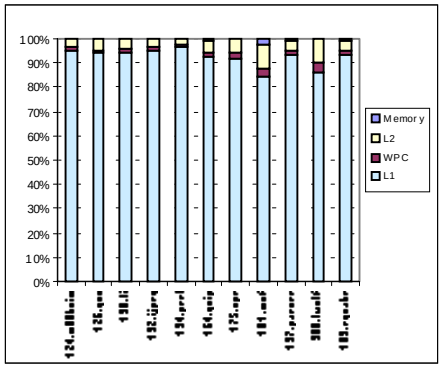

Fig.3. The fraction of memory references on the *correct* execution path that are serviced by the L1 cache, the WPC, the L2 cache, and memory. The L1 data cache is 8KB direct-mapped and has 32-byte blocks.

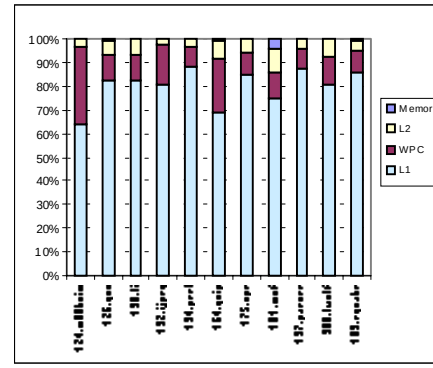

Fig 4. The fraction of memory references on the *wrong* execution path that are serviced by the L1 cache, the WPC, the L2 cache, and memory. The L1 data cache is 8KB direct-mapped and has 32-byte blocks.

cycles. For most of these programs, we see that the WPC converts approximately 20-35% of misses that would have been serviced by the L2 cache or the memory into accesses that are equivalent to an L1 hit.

While the above discussion explains some of the speedups seen in Figures 1 and 2, it does not completely explain the results. For instance, *twolf* has the largest fraction of memory accesses serviced by the WPC in Figure 3. However, *mcf*, *gzip*, and *equake* show better overall speedups. This difference in speedup is explained in Figure 4. This figure shows which levels of the memory hierarchy service the speculative loads issued on what is subsequently determined to be the wrong branch path. Speculative loads that miss in both the L1 cache and the WPC are serviced either by the L2 cache or by the memory. These values are placed in the WPC in the hope that the values will be subsequently referenced by a load issued on the correct branch path.

In Figure 4, we see that 30 percent of the wrong path accesses that miss in both the L1 and the WPC are serviced by memory, which means that this percentage of the blocks in the WPC are loaded from memory. So, from Figure 3 we can say that 30 percent of the correct path accesses that hit in the WPC for *mcf* would have been serviced by the memory in a system without the WPC. That is, the WPC effectively converts a large fraction of this program's L1 misses into the equivalent of an L1 hit. In *twolf*, on the other hand, most of the hits to the WPC would have been hits in the L2 cache in the absence of the WPC. We see in Figure 4 that less than 1% of the wrong path accesses for *twolf* that miss both in the L1 and the WPC are serviced by memory, while 99% of these misses are serviced by the L2 cache. That is, almost all the data in the WPC comes from the L2 cache for *twolf*. Thus, the WPC does a better job of hiding miss delays for *mcf* than for *twolf*, which explains why *mcf* obtains a higher overall speedup with the WPC than does *twolf*. A similar argument explains the speedup results observed in the remainder of the programs, as well.

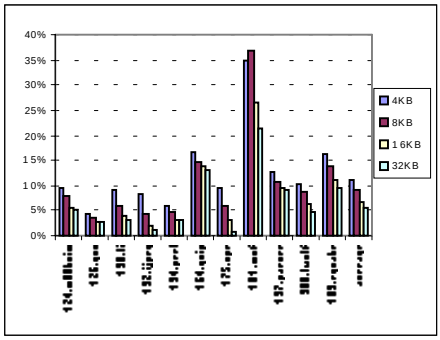

Fig. 5. Speedup obtained with the *wpc* configuration as the L1 cache size is varied. The L1 data cache is direct-mapped with 32-byte blocks. All speedups are relative to the baseline (*orig*)processor.

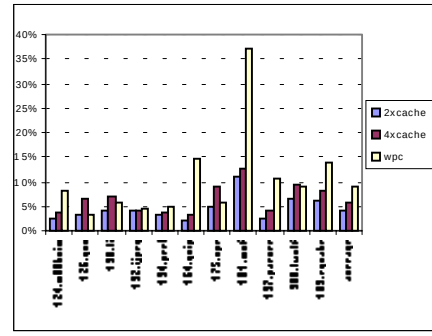

Fig. 6. The speedup obtained with the WPC compared to configurations with larger L1 caches but without a WPC. The base cache size is 8KB and is direct-mapped with 32- byte blocks.

4.2 Sensitivity to Cache Parameters

There are several parameters that affect the performance of a cache memory system. In this study, we examine the effects of the cache size, the associativity, and the cache block size on the cache performance when allowing the execution of wrong-path loads both with and without the WPC. Due to lack of space, the effects of memory latency and the size of WPC are not given in this paper. See [26] for information on the effects of these parameters.

Figure 5 shows that the relative benefit of the *wpc* decreases as the L1 cache size increases. However, the WPC size is kept constant in these simulations so that the relative size of the WPC to the data cache is reduced. With a smaller cache, wrong-path loads cause more misses compared to configurations with larger caches. These additional misses tend to prefetch data that is put into the WPC for use by subsequently executed correct branch paths. The WPC eliminates the pollution in the L1 data cache for the *all* configuration that would otherwise have occurred without the WPC, which then makes these indirect prefetches useful for the correct branch path execution.

While the WPC is a relatively small hardware structure, it does consume some chip area. Figure 6 shows the performance obtained with an 8-entry WPC used in conjunction with an 8KB L1 cache compared to the performance obtained with the original processor configuration using a 16KB L1 cache or a 32KB L1 cache but without a WPC. We find that, for all of the test programs, the small WPC with the 8KB cache exceeds the performance of the processor when the cache size is doubled, but without the WPC. Furthermore, the WPC configuration exceeds the performance obtained when the size of the L1 cache is quadrupled for all of the test programs except *gcc*, *li*, *vpr*, and *twolf*. We conclude that this small WPC is an excellent use of the chip area compared to simply increasing the L1 cache size.

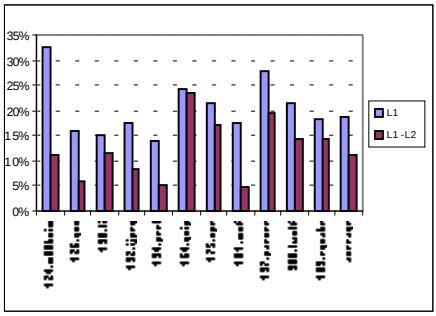

Fig. 7. The percentage increase in L1 cache accesses and traffic between the L1 cache and the L2 cache for the *wpc* configuration compared to the *orig* configuration. The L1 cache is 8 KB, direct-mapped and has 32-byte blocks.

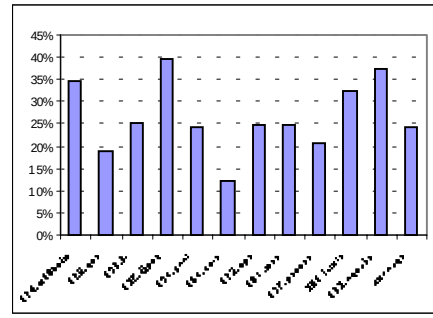

Fig. 8. The reduction in data cache misses for the *wpc* configuration compared to the *orig* configuration. The L1 cache is 8 KB, direct-mapped and has 32-byte blocks.

Figure 7 shows that executing the loads that are known to be down the wrong path typically increases the number of L1 data cache references by about 15-25% for most of the test programs. Furthermore, this figure shows that executing these wrong-path loads increases the bus traffic (measured in bytes) between the L1 cache and the L2 cache by 5-23%, with an average increase of 11%. However, the WPC reduces the total data cache miss ratio for loads on the correct path by up to 39%, as shown in Figure 8.

Increasing the L1 cache associativity typically tends to reduce the number of L1 misses on both the correct path [8] and the wrong path. This reduction in misses reduces the number of indirect prefetches issued from the wrong path, which then reduces the impact of the WPC, as shown in Figure 9. The *mcf* program is the exception since its overall cache behavior is less sensitive to the L1 associativity than the other test programs.

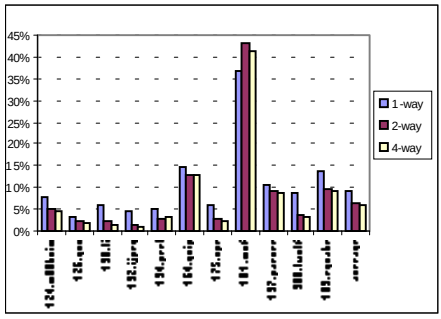

Fig. 9. The effect of the L1 cache associativity on the speedup of the *wpc* configuration compared to the *orig* configuration. The L1 cache size is 8 KB with 32-byte blocks.

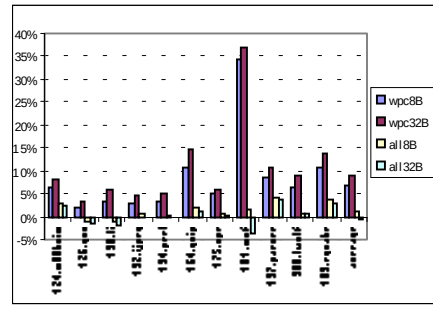

Fig. 10. The effect of the cache block size on the speedup of the *all* and *wpc* configurations compared to the *orig* configuration. The L1 cache is direct-mapped and 8 KB. The WPC is 256B, i.e, 8–entries with 32-byte blocks (wpc32B), or 32-entries with 8-byte blocks (wpc8B).

As the block size of the data cache increases, the number of conflict misses also tends to increase [8, 27]. Figure 10 shows that smaller cache blocks produce better speedups for configurations without a WPC when wrong-path loads are allowed to execute since larger blocks more often displace useful data in the L1 cache. However, for the systems with a WPC, the increasing conflict misses in the data cache due to the larger blocks increases the number of misses that hit in the WPC because of the victim-caching behavior of the WPC. In addition, the indirect prefetches provide a greater benefit for large blocks since the WPC eliminates their polluting effects. We conclude that larger cache blocks work well with the WPC since the strengths and weaknesses of larger blocks and the WPC are complementary.

5 Related Work

There have been several studies examining how speculation affects multiple issue processors [1-7]. Farkas *et al* [1], for example, looked at the relative memory system performance improvement available from techniques such as non-blocking loads, hardware prefetching, and speculative execution, used both individually and in combination. The effect of deep speculative execution on cache performance has been studied by Pierce and Mudge [2]. Several other authors [3-7] examined speculation and pre-execution in their studies Wallace et al. [4] introduced instruction recycling, where previously executed wrong path instructions are injected back into the rename stage instead of being discarded. This technique increases the supply of instructions to the execution pipeline and decreases fetch latency.

Prefetching, which overlaps processor computations with data accesses, has been shown to be one of several effective approaches that can be used to tolerate large memory latencies. Prefetching can be hardware-based, software-directed, or a combination of both [21]. Software prefetching relies on the compiler to perform static program analysis and to selectively insert prefetch instructions into the executable code [16-19]. Hardware-based prefetching, on the other hand, requires no compiler support, but it does require some additional hardware connected to the cache [8-15]. This type of prefetching is designed to be transparent to the processor.

Jouppi [9] proposed *victim caching* to tolerate conflict misses. While several other prefetching schemes have been proposed, such as adaptive sequential prefetching [11], prefetching with arbitrary strides [11, 14], fetch directed prefetching [13], and selective prefetching [15], Pierce and Mudge [20] have proposed a scheme called wrong path instruction prefetching. This mechanism combines next-line prefetching with the prefetching of all instructions that are the targets of branch instructions regardless of the predicted direction of conditional branches.

Most of the previous prefetching schemes require a significant amount of hardware to implement. For instance, they require a *prefetcher* that prefetches the contents of the missed address into the data cache or into an on-chip prefetch buffer. Furthermore, a *prefetch scheduler* is needed to determine the right time to prefetch. On the other hand, this work has shown that executing loads down the wrongly-predicted branch paths can provide a form of indirect prefetching, at the potential expense of some cache pollution. Our proposed Wrong Path Cache (WPC) is essentially a combination of a very small prefetch buffer and a victim cache [9] to eliminate this pollution effect.

6 Conclusions

This study examined the performance effects of executing the load instructions that are issued along the incorrectly predicted path of a conditional branch instruction. While executing these wrong-path loads increases the total number of memory references, we find that allowing these loads to continue executing, even after the branch is resolved, can reduce the number of misses observed on the correct branch path. Executing these wrong-path loads thus provides an indirect prefetching effect. For small caches, however, this prefetching can pollute the cache causing an overall slowdown in performance.

We proposed the Wrong Path Cache (WPC), which is a combination of a small prefetch buffer and a victim cache, to eliminate the pollution caused by the execution of the wrong-path loads. Simulation results show that, when using an 8 KB L1 data cache, the execution of wrong-path loads without the WPC can result in a speedup of up to 5%. Adding a fully-associative eight-entry WPC to an 8 KB direct-mapped L1 data cache, though, allows the execution of wrong path loads to produce speedups of 4% to 37% with an average speedup of 9%. The WPC also shows substantially higher speedups compared to the baseline processor equipped with a victim cache of the same size.

This study has shown that the execution of loads that are known to be from a mispredicted branch path has significant potential for improving the performance of aggressive processor designs. This effect is even more important as the disparity between the processor cycle time and the memory speed continues to increase. The Wrong Path Cache proposed in this paper is one possible structure for exploiting the potential benefits of executing wrong-path load instructions.

Acknowledgement

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

References

[1] K. I. Farkas, N. P. Jouppi, and P. Chow, "How Useful Are Non-Blocking Loads, Stream Buffers, and Speculative Execution in Multiple Issue Processors?" *Technical Report WRL RR 94/8*, Western Research Laboratory – Compaq, Palo Alto, CA, August 1994.

[2] 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.

[3] G. Reinman, T. Austin, and B. Calder, "A Scalable Front-End Architecture for Fast Instruction Delivery," *26th International Symposium on Computer Architecture*, pages 234-245, May 1999.

[4] S. Wallace, D. Tullsen, and B. Calder, "Instruction Recycling on a Multiple-Path Processor," *5th International Symposium On High Performance Computer Architecture*, pages 44-53, January 1999.

[5] G. Reinman and B. Calder, "Predictive Techniques for Aggressive Load Speculation," *31st International Symposium on Microarchitecture*, pages 127-137, December 1998.

[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," In *28th International Symposium on Computer Architecture*, July, 2001.

[7] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," *Proc. 1997 ACM Int. Conf. on Supercomputing*, July 1997, pp. 68-75.

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

[9] 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.

[10] F. Dahlgren, M. Dubois and P. Stenstrom, "Fixed and Adaptive Sequential Prefetching in Shared-memory Multiprocessors," *Proc. First IEEE Symposium on High Performance Computer Architecture*, Raleigh, NC, Jan. 1995, pp. 68-77.

[11] T.F. Chen and J.L Baer, "Effective Hardware-Based Data Prefetching for High Performance Processors," *IEEE Transactions on Computers*, Vol. 44, No.5, May 1995, pp. 609-623.

[12] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *IEEE Transactions on Computers*, Vol. 48, No 2, 1999, pp 121-133.

[13] G. Reinman, B. Calder, and T. Austin, "Fetch Directed Instruction Prefetching," *In proceedings of the 32nd International Symposium on Microarchitecture*, November 1999.

[14] T.F. Chen and J.L Baer, "A Performance Study of Software and Hardware Data Prefetching Schemes," *Proc. of the 21st Annual International Symposium on Computer Architecture*, Chicago, Il, April 1994, pp. 223-234.

[15] R. Pendse and H. Katta, "Selective Prefetching: Prefetching when only required," *Proc. of the 42nd IEEE Midwest Symposium on Circuits and Systems*, volume 2, 2000, pp. 866-869.

[16] C-K. Luk and T. C. Mowry. "Compiler-based prefetching for recursive data structures," *In Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222--233, Oct. 1996.

[17] Bernstein, D., C. Doron and A. Freund, "Compiler Techniques for Data Prefetching on the PowerPC," *Proc. International Conf. on Parallel Architectures and Compilation Techniques*, June 1995, pp. 19-26.

[18] E.H. Gornish, E.D. Granston and A.V. Veidenbaum, "Compiler-directed Data Prefetching in Multiprocessors with Memory Hierarchies," *Proc. 1990 International Conference on Supercomputing*, Amsterdam, Netherlands, June 1990, pp. 354-368.

[19] M.H. Lipasti, W.J. Schmidt, S.R. Kunkel and R.R. Roediger, "SPAID: Software Prefetching in Pointer and Call-Intensive Environments," *Proc. 28th Annual International Symposium on Microarchitecture*, Ann Arbor, MI, November 1995, pp. 231-236.

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

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

[22] 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.

[23] AJ KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja, "Adapting the SPEC 2000 Benchmark Suite for Simulation-Based Computer Architecture Research," *Workload Characterization of Emerging Computer Applications*, L. Kurian John and A. M. Grizzaffi Maynard (eds.), Kluwer Academic Publishers, pp 83-100, (2001).

[24] S-T Pan, K. So, and J.T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," *Proc. of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992, pp. 76-84.

[25] T.Y. Yeh and Y. N. Patt, "A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History," *Proc. of the International Symposium on Computer Architecture*, 1993, pp. 257--267.

[26] R. Sendag, D. J. Lilja, and S. R. Kunkel, "Exploiting the Prefetching Effect provided by Executing Mispredicted Load Instructions," *Laboratory for Advanced Research in Computing Technology and Compilers, Technical Report No. ARCTIC 02-05, May 2002.*

[27] D. A. Patterson and J. L. Hennessy: *Computer Architecture: A Quantitative Approach*, 2nd edition, Morgan Kaufmann press, 1995, pp. 393-395.