Cache Filtering Techniques to Reduce the Negative Impact of Useless Speculative Memory References on Processor Performance

Onur Mutlu Hyesoon Kim David N. Armstrong Yale N. Patt

Department of Electrical and Computer Engineering
The University of Texas at Austin
{onur,hyesoon,dna,patt}@ece.utexas.edu

Abstract

High-performance processors employ aggressive speculation and prefetching techniques to increase performance. Speculative memory references caused by these techniques sometimes bring data into the caches that are not needed by correct execution. This paper proposes the use of the first-level caches as filters that predict the usefulness of speculative memory references. With the proposed technique, speculative memory references bring data only into the first-level caches rather than all levels in the cache hierarchy. The processor monitors the use of the cache blocks in the first-level caches and decides which blocks to keep in the cache hierarchy based on the usefulness of cache blocks. It is shown that a simple implementation of this technique usually outperforms inclusive and exclusive baseline cache hierarchies commonly used by today's processors and results in IPC performance improvements of up to 9.2% on the SPEC2000 integer benchmarks.

1. Introduction

Branch prediction and prefetching are two of the most effective techniques used by processors to achieve high performance. When accurate, these techniques improve performance significantly. However, incorrect branch predictions and inaccurate prefetch requests result in memory references that bring data into the caches that are not needed by correct-path execution. We name these references "useless speculative memory references." Useless speculative memory references may be detrimental to processor performance in two major ways: they may cause cache pollution by evicting cache blocks that will be used by correct-path execution and they may consume bandwidth and resources that are needed by correct-path memory references.

In this paper, we show that the dominant negative effect of speculative memory references is cache pollution, in particular second-level cache pollution. Based on this observation and an analysis of the behavior of speculative memory references, we propose a simple, novel cache filtering technique to reduce the second-level cache pollution caused by useless speculative memory references. We evaluate the performance of the proposed mechanism on several aggressive machine models and show that it usually outperforms the baseline cache hierarchies commonly used by today's processors which do not employ filtering.

2. Motivation

Useless speculative memory references are detrimental to processor performance because they cause cache pollution and consume bandwidth and resources. In this section we quantify the performance impact of cache pollution as well as unnecessary bandwidth and resource usage of speculative memory references. The baseline processor we use for the experiments in this section uses an aggressive branch predictor and an aggressive stream-based prefetcher[1].

To quantify the performance impact of cache pollution and bandwidth and resource usage caused by speculative memory references, we simulate three idealized processor models and compare the performance of these models to that of the baseline processor. The first idealized model is a processor in which speculative memory references do not cause any pollution in the first-level (L1) caches. In the second model, speculative memory references do not cause pollution in the second-level (L2) cache[2]. In both of these models, the bandwidth and resources used by speculative memory references are realistically modeled. In the third model, speculative memory references never get in the way of non-speculative references by consuming bandwidth or

[1]The baseline processor is described in Section 5.

[2]To eliminate pollution in both of these models, speculative memory references resulting from memory accesses on the wrong path as well as prefetch requests, are fetched into a separate idealized buffer and moved into the caches if they are used by correct-path execution.

resources, but they can cause cache pollution. The performance improvement of these three idealized models compared to the baseline model for SPEC2000 integer benchmarks is shown in Figure 1.

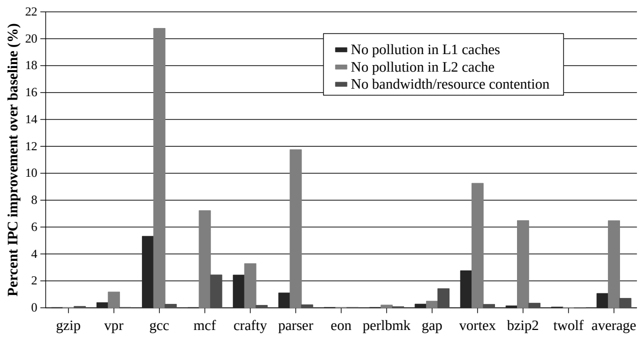

Figure 1. IPC improvement over the baseline processor with stream prefetching (*stream-baseline* described in Section 5) if negative effects of speculative memory references (L1 cache pollution, L2 cache pollution, bandwidth/resource contention) are eliminated.

Figure 1 shows that the negative performance impact of speculative memory references is primarily due to L2 cache pollution. Therefore, techniques to reduce L2 cache pollution caused by speculative memory references have the potential to increase processor performance. Based on this observation, we present a technique that aims to increase performance by reducing the L2 cache pollution caused by speculative memory references.

3. Analysis of Speculative References

In order to design a mechanism to eliminate pollution due to speculative memory references, we need to understand the behavior of these references. In this section, we examine *when*, if at all, these references are used after they are brought into the cache hierarchy.

Figure 2 shows the distribution of the time between the fetch and correct-path use of speculatively-fetched (by wrong-path references or the prefetcher) data cache blocks for the gcc benchmark. The X-axis, which is in log-scale, represents the number of cycles between the placement of a block into the caches and the use of that block by a correct-path instruction. The Y-axis shows the number of speculatively-fetched data cache blocks. The rightmost bar shows the number of speculatively-fetched blocks that are never used by correct-path instructions during the entire execution of the benchmark. The average time a block stays in the data cache (the average D-cache residence time, calculated over *all* blocks evicted from the data cache) and the average time a block stays in the L2 cache are also shown on the graph. Note that the data in Figure 2 are collected by monitoring the use of the speculatively-fetched blocks even after they are evicted from all caches. Therefore, a block that is requested by a correct-path instruction, even after the block was evicted from the caches is counted as "used."

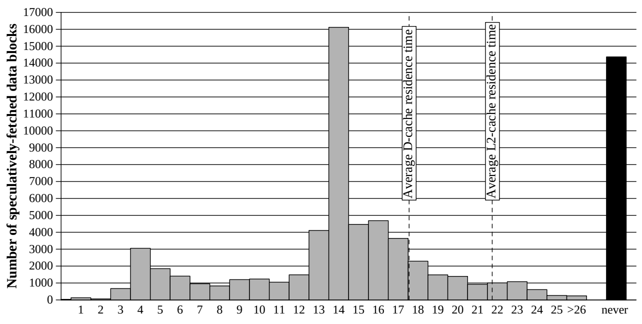

Figure 2. Distribution of the time (in number of cycles) between the fetch and correct-path use of speculatively-fetched data cache blocks for the gcc benchmark. X-axis is in log-scale (base 2).

Figure 2 shows that most speculatively-fetched blocks are used before the average D-cache residence time, while they are still likely to be in the data cache. If a speculatively-fetched block is not used before the average D-cache residence time, then it is unlikely that the block will be used before the average L2 residence time, if it is used at all. That is, the total number of blocks to the right of the average L2 cache residence time, including the blocks that are never used, is greater than the total number of blocks between the average D-cache residence time and the average L2 cache residence time in Figure 2. Therefore, if a speculatively-fetched block is going to be used, then it is most likely that it will be used while it resides in the data cache. This observation provides the motivation for using the data cache as a filter for speculative memory references.

Figure 3 shows the percentage breakdown of all speculatively-fetched data cache blocks for the six SPEC2000 integer benchmarks that suffer from L2 cache pollution[3]. The speculatively-fetched blocks are broken down into four categories: blocks used before the average D-cache residence time, blocks used between the average D-cache residence time and the average L2 cache residence time, blocks used after the average L2 cache residence time, and blocks that are never used. According to this figure, all the benchmarks except for bzip2, exhibit a behavior similar to that which was observed for gcc in Figure 2. In every case, more then 65% of the speculatively-fetched blocks are used before the average D-cache residence time. The percentage of blocks that are never used and the percentage of blocks that are used after the average L2 cache

[3]These six benchmarks are benchmarks that gain at least 2% IPC improvement from eliminating L2 cache pollution in Figure 1.

residence time added together significantly exceed the percentage of blocks that are used after the average D-cache residence time, but before the average L2 cache residence time. Only in bzip2 is this behavior not observed. We find that speculative instruction references also show a similar behavior[4].

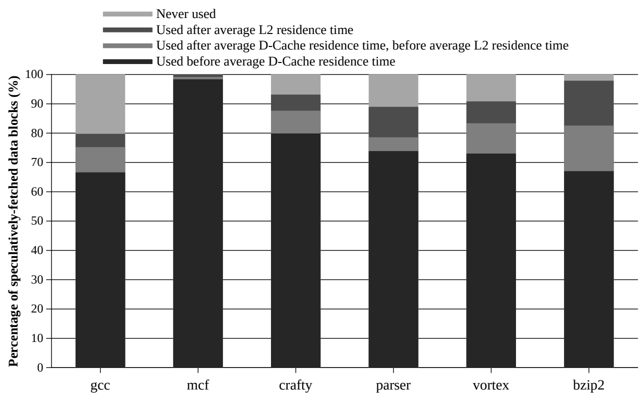

Figure 3. Breakdown of speculatively-fetched data cache blocks based on their use time.

3.1. Wrong-path vs. Prefetcher References

We investigate the behavior of wrong-path references and prefetcher references separately. Figure 4 shows the breakdown of speculatively-fetched data cache blocks based on their use time for five benchmarks[5]. The left bar for each benchmark shows the number of data cache blocks fetched by wrong-path references. The right bar shows the number of blocks fetched by the prefetcher. For all five benchmarks, the prefetcher fetches more blocks than the wrong-path references. In general, the blocks fetched by wrong-path references are more likely to be used by correct-path execution and they are more likely to be used *early* (before the average D-cache residence time) compared to the blocks fetched by the prefetcher. This is true especially for bzip2. In bzip2, there is a very large number of blocks fetched by the prefetcher that are used between the average D-cache residence time and the average L2 residence time. Hence, for bzip2, if a prefetched cache block is not used before the average D-cache residence time, it is still likely to be used before the average L2 residence time. Therefore, using the data cache as a filter for prefetch requests may not be effective in bzip2.

[4]We do not present these results due to space constraints. Only gcc and crafty have a significant number of speculatively-fetched instruction cache blocks.

[5]Mcf is omitted from Figure 4 because almost all speculatively-fetched blocks are used before the average D-cache residence time regardless of whether they are fetched by the wrong path or the prefetcher.

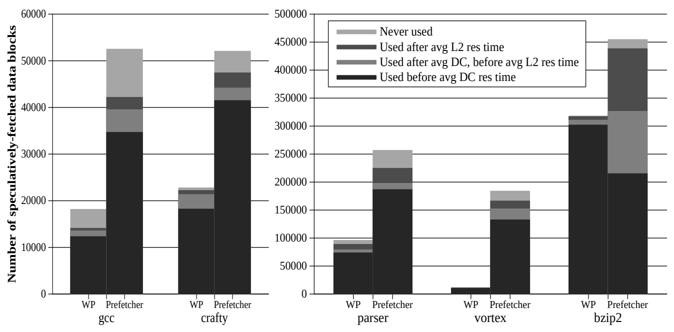

Figure 4. Breakdown of data blocks fetched by the wrong path (WP) and the prefetcher based on their use time.

4. Cache Filtering Techniques

In the previous section, we have shown that most benchmarks that suffer from L2 cache pollution due to speculative memory references conform to the following characteristic: “If a speculatively-fetched cache block is not used while it resides in the first-level cache, then it is likely that the block will not be used at all, or will not be used before it is evicted from the L2 cache.” Based on this characteristic, in this section, we propose filtering techniques to reduce the L2 cache pollution caused by speculative memory references.

4.1. First-Level Caches as Filters

We propose a mechanism wherein the first-level caches are used to predict the usefulness of speculatively-fetched cache blocks and to filter out useless speculatively-fetched blocks. In this mechanism, all memory references made by wrong-path instructions or the prefetcher are fetched *only* into the first-level cache, instead of into both the first-level and second-level caches[6]. While the speculatively-fetched blocks reside in the first-level cache, the processor monitors whether they are referenced by non-speculative (correct-path) instructions.

If a speculatively-fetched block is referenced by a non-speculative instruction while it resides in the first-level cache, then the speculatively fetched block is treated as any other non-speculatively-fetched block. When the block is replaced from the first-level cache, it is written into the second-level cache.

If a speculatively-fetched block is not referenced by a non-speculative instruction before it is evicted from the first-level cache, the processor predicts that the block will never be used or it will be replaced from the L2 cache before it is used. Based on this prediction, the processor may choose to *not* write the block into the L2 cache or may adopt a policy that gives lower priority to the unused

[6]In-flight speculative references that are also requested by correct-path instructions are considered non-speculative.

speculatively-fetched block. We discuss some simple policies in the next section.

4.2. Filtering Policies

If a speculatively-fetched block is not referenced by a non-speculative instruction before it is replaced from the first-level cache, what should the processor do with the block when it is evicted from the data cache? We propose two simple policies to answer this question.

One policy is not to write the block into the L2 cache. We call this policy *no-spec-L2fill* (i.e., no speculative L2 fill). This policy prevents all the useless speculative references from polluting the L2 cache. Unfortunately, such a policy also filters out some useful speculatively-fetched blocks that would have been used if they were placed into the L2 cache.

The second policy, called *spec-L2fill-lru*, is to write the block into the L2 cache, but to write it into the LRU (least recently used) slot of its set. This policy captures the benefit of those speculatively-fetched blocks that are referenced by correct-path execution shortly after they are replaced from the first-level cache. Useless speculatively-fetched blocks may still evict useful L2 cache blocks, but the effect is less pronounced because useless blocks occupy an L2 cache entry for a shorter amount of time.

More complicated filtering policies, such as complex predictors to predict the usefulness of speculatively-fetched blocks, can be implemented at the expense of simplicity. Such predictors are out of the scope of this paper and part of our future work.

4.3. Implementation Considerations

The proposed techniques require that the processor distinguish speculative memory references from non-speculative memory references. This is easily done for hardware prefetcher requests, but it requires more effort for wrong-path requests. An implementation can, and our simulations do, assume that every request is non-speculative until it is known to be speculative. Until an older mispredicted branch is resolved, a memory reference made by a wrong-path instruction is considered non-speculative. Such an implementation would probably bring in some unused speculative cache blocks into especially the first-level caches. We find that mispredicted branches are resolved before 94% of wrong-path L2 misses complete. Therefore, whether an L2 cache miss is speculative is usually known before the block is placed into the L2 cache. A processor implementing the proposed filtering techniques will place most of the blocks fetched by wrong-path L2 cache misses only into the first-level cache.

Most processors use a buffer, such as the MSHRs [9], to keep track of the outstanding memory requests. This buffer can be augmented to include the sequence number[7] of the oldest instruction that requires the data of the request. When a mispredicted branch is resolved, the processor can compare the sequence number associated with each entry to the sequence number of the branch and mark the requests with larger sequence numbers as speculative in the memory request buffer. This requires one bit per entry in the memory request buffer. All prefetcher requests are initially marked as speculative. If a non-speculative instruction requests a block that is already in flight, the speculative bit associated with the request is reset.

To mark the blocks as speculative or non-speculative, the tag arrays of the first-level caches require one bit per entry[8]. The speculative bit is set if the block placed into the cache was marked speculative in the memory request buffer. At retirement time, instructions reset the speculative bits of the first-level cache blocks they access.

5. Experimental Methodology

We evaluate the proposed techniques on a simulator that can simulate Alpha ISA binaries. We use an execution-driven simulator capable of accurately modeling wrong-path execution. The processor we model is an aggressive superscalar, out-of-order processor. We evaluate two baseline processors: one that uses an aggressive stream-based prefetcher (*stream-baseline*) and another that employs runahead execution [14] (*runahead-baseline*). The configuration common to both baselines is shown in Table 1.

Table 1. Baseline processor configuration.

Fetch/Issue/Retire width	8 instructions/cycle, 8 functional units
Pipeline length	20 stages
Instruction window size	128-entry instruction window, 128-entry ld-st queue
Cond. branch predictor	64K-entry gshare, 64K-entry PAs [18] hybrid
Indirect predictors	64K-entry, 4-way [3], 32-entry return address stack
L1 Instruction Cache	64KB, 4-way, LRU replacement
L1 Data Cache	64KB, 4-way, 8 banks, 2-cycle, LRU repl.
L2 Unifi ed Cache	512KB, 8-way, 8 banks, 12-cycle, LRU repl.
Cache block size	64 bytes for all caches
Memory request buffers	128 buffers that hold L1 and L2 miss requests
Processor-memory bus	32-byte, split-transaction, 100-cycle one-way latency
Main memory	32 banks, 300-cycle bank access latency

Bandwidth, port contention, bank conflicts, and queuing effects are modeled at all levels in the memory hierarchy. L1 and L2 caches are inclusive. *Stream-baseline* employs an aggressive stream-based prefetcher similar to the one described in [16] that can detect and generate prefetches for 16 different streams. The prefetcher prefetches into the L2 cache. Memory system gives priority to load and store instruction requests over prefetcher requests.

Runahead-baseline employs runahead execution (a method of speculative pre-execution triggered when a long-

[7]e.g. the inums used in Alpha 21264 [7]

[8]Physically, these bits can be separate from the tag array.

latency L2 miss instruction blocks retirement from the instruction window) as a means to achieve the performance of a larger instruction window [14]. Requests generated during runahead mode are considered speculative. No prefetcher exists in the *runahead-baseline*. We use the *runahead-baseline* to evaluate the performance of our filtering techniques in the presence of very aggressive speculation without stream prefetching. On this baseline, the proposed techniques eliminate pollution only due to wrong-path references.

The experiments were run using the 12 SPEC2000 integer benchmarks compiled for the Alpha ISA with the `-fast` optimizations and profiling feedback enabled. The benchmarks were run to completion with a reduced input set [8] to reduce simulation time.

Figure 5 shows the IPC (retired Instructions Per Cycle) performance of both baseline processors. The IPC of a processor that employs neither stream-prefetching nor runahead execution is also shown for reference. We note that both stream prefetching and runahead execution are very effective in improving IPC. We evaluate the proposed techniques on the *stream-baseline* and *runahead-baseline* assuming that high-performance processors will employ either method. Unless otherwise noted, all IPC results reported in this paper are relative to the corresponding baseline in Figure 5. Average IPC values are calculated as the harmonic mean of the IPC values for benchmarks.

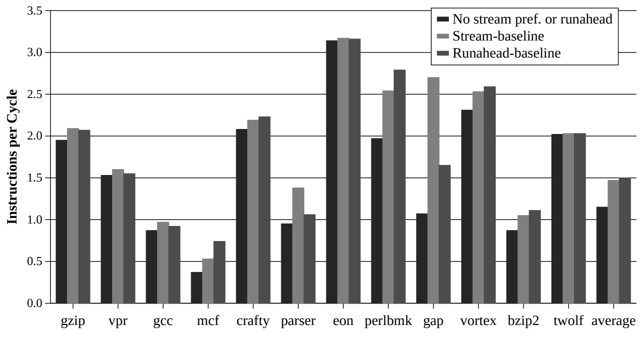

Figure 5. Baseline IPC performance.

6. Experimental Results

Figure 6 shows the percent change in IPC compared to the *stream-baseline* when the filtering techniques discussed in Section 4.2 are applied. *no-spec-L2fill* policy reduces performance significantly on gap and bzip2, because these two benchmarks have many speculatively-fetched cache blocks that are needed after they are evicted from the first-level caches, as explained in Section 3.1. *spec-L2fill-lru* policy recovers most of the performance loss incurred by these two benchmarks. Gcc, parser, and vortex, three benchmarks which suffer the most from L2 cache pollution as was shown in Figure 1, benefit significantly from both policies with a maximum performance improvement of 4.7% for parser. These results show that using the first-level caches as filters improves performance and *spec-L2fill-lru* is an effective filtering policy.

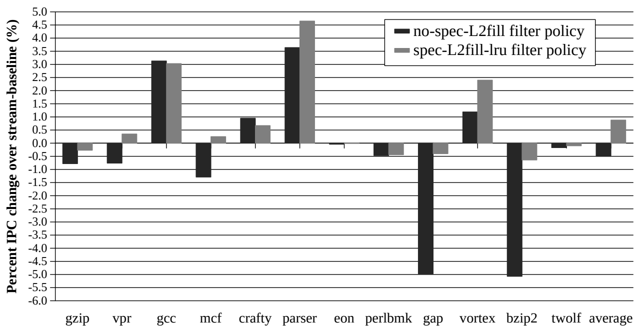

Figure 6. Performance improvement of filtering mechanisms on the *stream-baseline*.

Figure 7 shows the percent change in IPC compared to the *runahead-baseline* when the two filtering techniques are applied. The maximum performance improvement is 9.2% for mcf. A performance degradation is observed only for bzip2 (-0.7%). In mcf, runahead execution results in many wrong-path requests that are either never needed or needed by correct-path execution after being evicted from caches. The proposed filtering mechanism is very effective in filtering out these speculatively-fetched blocks.

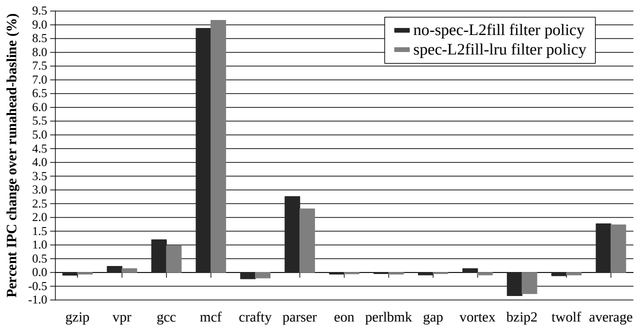

Figure 7. Performance improvement of filtering mechanisms on the *runahead-baseline*.

In contrast to the *stream-baseline*, the performance improvements corresponding to the two filtering policies are quite similar on the *runahead-baseline*. We find this is due to two reasons:

1. If wrong-path requests are not used while they are in the first-level cache, they are more likely to be never used than prefetcher requests. This behavior is also the reason why performance improvement of the filtering mechanism on the *runahead-baseline* is more posi-

tive than the performance improvement on the *stream-baseline*.

2. Runahead execution is better able to tolerate the extra misses caused by the *no-spec-L2fill* filtering policy.

6.1. Filtering in Exclusive Cache Hierarchies

The proposed filtering techniques violate the inclusion property [1] in the cache hierarchy. Although non-inclusive hierarchies can be implemented, we would also like to know the performance of these techniques on exclusive hierarchies [6]. An exclusive hierarchy can tolerate pollution better than an inclusive hierarchy, because exclusion increases the total effective cache size by eliminating redundancy in caches. Therefore, we hypothesize that filtering would be less effective on exclusive hierarchies.

Figure 8 shows the percent change in IPC compared to the *stream-baseline* of three processors: one with an exclusive hierarchy, one with an exclusive hierarchy that employs the *no-spec-L2fill* policy, and one with an exclusive hierarchy that employs the *spec-L2fill-lru* policy. Note that the exclusive hierarchy outperforms the inclusive hierarchy by more than 2% on the six benchmarks that suffer from L2 cache pollution, confirming the intuition that an exclusive hierarchy is more tolerant to pollution than an inclusive one. The *spec-L2fill-lru* filtering policy improves the performance of the exclusive hierarchy for vpr, gcc, crafty, and parser, but it reduces performance for mcf, gap, and bzip2. As hypothesized, filtering is less effective on the exclusive hierarchy that is more tolerant to pollution. In bzip2 and gap, many prefetched cache blocks are used after they are evicted from the data cache. Therefore, using the data cache to filter useless prefetch requests is not effective for these two benchmarks in either inclusive or exclusive hierarchies.

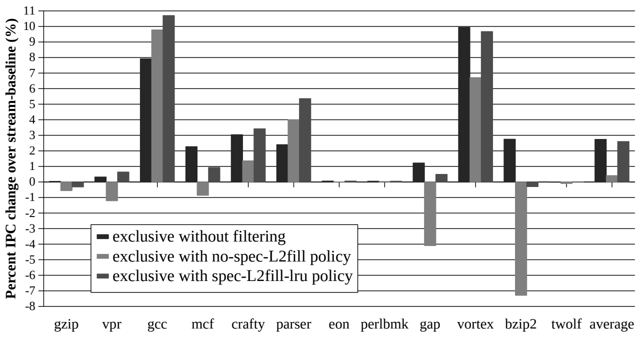

Figure 8. Performance improvement of exclusive caching and filtering mechanisms on exclusive caching over the *stream-baseline***.**

Figure 9 shows the percent change in IPC compared to the *runahead-baseline* of the corresponding four processors that employ runahead execution. Filtering on runahead processors with exclusive caches is generally beneficial for performance, especially for gcc and parser. This shows that filtering techniques are more successful for speculative wrong-path requests than they are for prefetch requests, even on exclusive cache hierarchies.

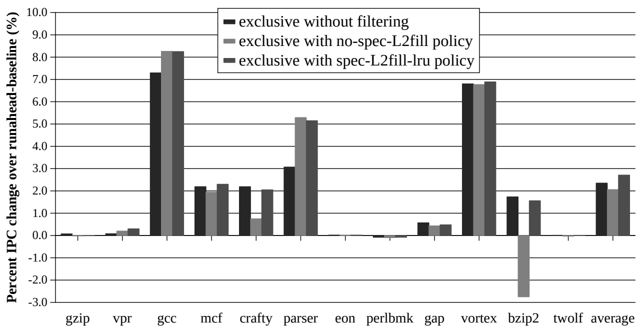

Figure 9. Performance improvement of exclusive caching and filtering mechanisms on exclusive caching over the *runahead-baseline***.**

6.2. Sensitivity to L1 Cache Size

The proposed techniques use the first-level caches as filters. Therefore, the effectiveness of these techniques depend on the sizes of the first-level caches. We would expect the filtering techniques to be more successful on processors with larger L1 caches, because the proposed techniques rely on monitoring the speculatively-fetched cache blocks while they reside in the L1 caches. Increasing the L1 cache size increases the time these blocks stay in the L1 cache, hence, gives more time to the processor to evaluate the usefulness of a speculatively-fetched block and make more accurate filtering decisions.

To determine the sensitivity of performance to the L1 cache size, we evaluate the effect of using *spec-L2fill-lru* policy on processors with four different L1 (data and instruction) cache sizes: 16KB, 32KB, 64KB, and 128KB. All other parameters of these processors are the same as the baseline described in Section 5.

Figure 10 shows the percent change in IPC compared to the *stream-baseline* with the respective L1 cache size when the *spec-L2fill-lru* filtering technique is applied. Even with 32KB L1 caches, more than 1% IPC improvement is observed for gcc, parser, and vortex.

Figure 11 shows the percent change in IPC compared to the *runahead-baseline* with the respective L1 cache size when the *spec-L2fill-lru* filtering technique is applied. Runahead processors also show the same trend of increased IPC improvement provided by filtering with increased L1 cache sizes. Filtering techniques improve average IPC even on runahead processors with small (16KB) L1 caches.

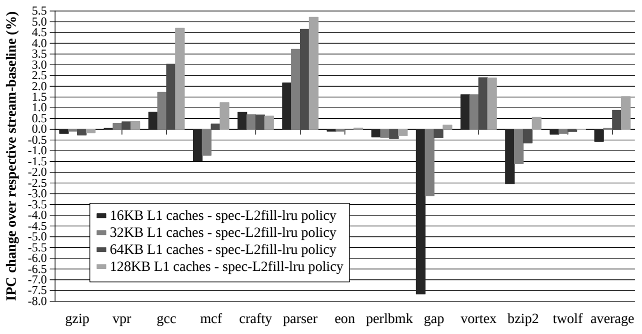

Figure 10. Performance improvement of the *spec-L2fill-lru* **filtering policy on four** *stream-baseline* **processors with different L1 cache sizes.**

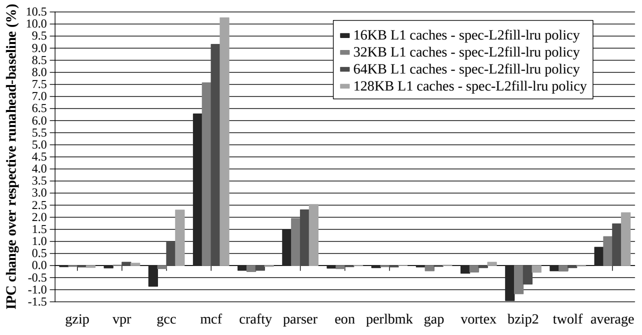

Figure 11. Performance improvement of the *spec-L2fill-lru* **filtering policy on four** *runahead-baseline* **processors with different L1 cache sizes.**

7. Related Work

7.1. Reducing the Negative Effects of Prefetching

Zhuang and Lee propose a filtering mechanism to reduce the cache pollution caused by useless prefetches [19]. This mechanism predicts the usefulness of a prefetch based on past history. Prefetches predicted to be useless are never requested from the memory system, thereby reducing L1 pollution and bandwidth contention caused by prefetches. Srinivasan et. al. propose a static filter to reduce the number of useless prefetch requests that consume bandwidth [15]. In their approach, the compiler identifies which load instructions trigger prefetch requests based on profile information. Our filtering mechanism differs from these approaches in two aspects:

1. We apply filtering *after* prefetches bring data into the L1 cache, not *before* requesting a prefetch. This choice is based on the observations that bandwidth/resource contention and L1 cache pollution caused by prefetches are much less detrimental to performance than L2 cache pollution.
2. Our mechanism is a general approach that targets all speculative memory references, including wrong-path references and prefetches.

Lai et. al. introduce a predictor that predicts that a cache block is dead (i.e. not going to be needed by the processor) [10]. Prefetched cache blocks replace those blocks that are predicted to be dead. If the prediction is correct, pollution caused by useless prefetches is reduced. This technique is orthogonal to the filtering technique we propose and both techniques can be combined for increased performance. For example, a filtering policy can use dead block prediction to decide which block in the L2 cache should be evicted when an unused speculatively-fetched first-level cache block needs to be written into the L2 cache.

Software schemes have been proposed to reduce cache pollution caused by prefetching by improving the cache replacement decisions [4, 17]. These techniques can also be combined with our technique for improved performance.

A prefetch buffer [11] eliminates cache pollution caused by prefetching. Prefetches are placed into the prefetch buffer instead of the caches. Unfortunately, adding a prefetch buffer increases the design complexity of the memory system. A prefetch buffer that provides good performance may require a large size in the presence of aggressive prefetching. Our purpose in this paper is to improve the performance of prefetching into the caches without significantly increasing the complexity of the memory system.

7.2. Reducing the Negative Effects of Wrong Path

Bahar and Albera [2] claim that data fetched by wrong-path memory references are likely to pollute the caches. In order to alleviate this pollution, they introduce a small fully-associative structure, accessed in parallel with the L1 data cache. Data blocks fetched by instructions that are predicted to be on the wrong path using a confidence predictor are placed into this structure instead of the L1 data cache. The performance improvement provided by this mechanism comes partly from the additional associativity provided by the separate structure.

Mutlu et al. analyze the effects of wrong-path memory references on processor performance [13]. They find that L2 cache pollution caused by useless wrong-path references is sometimes detrimental for performance. Our work builds on this work by proposing a mechanism to reduce the L2 cache pollution caused by wrong-path references as well as prefetcher references.

7.3. Related Cache Management Techniques

Johnson and Hwu present a heuristic to prevent the most frequently used cache blocks from being evicted out of the first-level data cache [5]. Their mechanism keeps track of the number of accesses to individual memory blocks and when the number of accesses to the missed block is less than some constant times the number of accesses to the block that would be evicted, the miss data is serviced directly to the register file, bypassing the cache.

Mekhiel proposes a scheme wherein a miss to memory is serviced to the second-level cache and must be accessed a second time before the block is moved to the first-level cache [12]. This scheme is intended to prevent an infrequently used block from replacing a frequently used block in the first-level cache. We find that, in light of longer memory latencies , it is more important to protect the contents of the second-level cache from pollution than it is to protect the first-level cache.

8. Conclusion and Future Work

This paper makes two main contributions:

1. It shows that the dominant negative effect of speculative memory references is the pollution they cause in the L2 cache.

2. It proposes a novel and simple technique that uses the first-level caches as filters to filter out useless speculative memory references and thus reduce the L2 cache pollution caused by them.

Future work in reducing the negative performance impact of speculative memory references includes the development of more accurate filtering policies. Filtering mechanisms on exclusive hierarchies also warrant more analysis.

9. References

[1] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 73–80, 1988.

[2] R. I. Bahar and G. Albera. Performance analysis of wrong-path data cache accesses. In *Workshop on Performance Analysis and its Impact on Design*, 1998.

[3] P.-Y. Chang, E. Hao, and Y. N. Patt. Predicting indirect jumps using a target cache. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 274–283, 1997.

[4] P. Jain, S. Devadas, and L. Rudolph. Controlling cache pollution in prefetching with software-assisted cache replacement. Technical Report CSG-462, Massachusetts Institute of Technology, 2001.

[5] T. L. Johnson and W. W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 315 – 326, 1997.

[6] N. P. Jouppi and S. J. E. Wilton. Tradeoffs in two-level on-chip caching. In *Proceedings of the 21st Intl. Symposium on Computer Architecture*, pages 34–45, 1994.

[7] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[8] A. KleinOsowski and D. J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.

[9] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Intl. Symposium on Computer Architecture*, pages 81–87, 1981.

[10] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction and dead-block correlating prefetchers. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 144–154, 2001.

[11] R. L. Lee, P.-C. Yew, and D. H. Lawrie. Data prefetching in shared memory multiprocessors. In *Proceedings of the Intl. Conference on Parallel Processing*, 1987.

[12] N. N. Mekhiel. Multi-level cache with most frequently used policy: A new concept in cache design. In *International Conference on Computer Applications in Industry and Engineering*, Nov 1995.

[13] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt. Understanding the effects of wrong-path memory references on processor performance. In *Third Workshop on Memory Performance Issues*, 2004.

[14] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th Intl. Symposium on High Performance Computer Architecture*, pages 129–140, 2003.

[15] V. Srinivasan, G. S. Tyson, and E. S. Davidson. A static filter for reducing prefetch traffic. Technical Report CSE-TR-400-99, University of Michigan Technical Report, 1999.

[16] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Technical White Paper*, Oct. 2001.

[17] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *Proceedings of the 12th Intl. Conference on Parallel Architectures and Compilation Techniques*, 2002.

[18] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 124–134, 1992.

[19] X. Zhuang and H.-H. S. Lee. A hardware-based cache pollution filtering mechanism for aggressive prefetches. In *Proceedings of the 32nd Intl. Conference on Parallel Processing*, pages 286–293, 2003.