On the Importance of Optimizing the Configuration of Stream Prefetchers

Ilya Ganusov and Martin Burtscher
Computer Systems Laboratory, Cornell University
Ithaca, New York
{ilya, burtscher}@csl.cornell.edu

ABSTRACT

This paper provides a detailed analysis of how the parameters of hardware prefetchers affect the memory system performance. In particular, we found the configuration of the frequently used stream prefetcher to have a major impact on the runtime, making parameter optimizations imperative when comparing a stream prefetcher with other prefetching techniques. For example, we show that adjusting the prefetch distance to the optimal value can increase the average speedup over the SPECcpu2000 benchmark suite from 40% to 70%. Moreover, our investigation of the performance of runahead prefetching relative to stream prefetching shows that choosing a non-optimal stream prefetcher as a baseline can distort the results by as much as a factor of two.

Categories and Subject Descriptors

B.8 [**Performance and Reliability**]: Miscellaneous

General Terms

Performance

Keywords

hardware prefetching, stream prefetcher, runahead execution

1. INTRODUCTION

The cores of modern high-end microprocessors deliver only a fraction of their theoretical peak performance. One of the main reasons for this inefficiency is the long latency of memory accesses. Often, load instructions that miss in the on-chip caches reach the head of the reorder buffer before the data is received, which clogs the pipeline and stalls the processor. As a result, the number of instructions executed per unit time is much lower than what the CPU is capable of handling. The rapid improvements in processor clock speed exacerbate this problem by widening the gap between the speed of the processor and the main memory.

Memory prefetching attempts to guess what data the program will need in the future and reads them in advance of the actual program references. If the prefetch is initiated sufficiently far in advance of the load, the entire latency of the memory access is hidden and the load does not result in any stall cycles. Even if the prefetch does not hide all of the access latency, the prefetch is still beneficial because it reduces the cycles that the processor is stalled waiting for the load to complete.

Many general-purpose processors have instruction set support for software prefetching. The processor moves the cache line addressed by software prefetch instructions to a specified level in the cache hierarchy. Software prefetching schemes have proven to be very effective for regular applications, which are dominated by loops traversing regular data structures such as arrays. Optimizing compilers are generally successful at generating binaries with software prefetching that significantly reduce the number of critical cache misses [12, 9].

Hardware prefetchers observe the behavior of a running application and initiate prefetching on repetitive patterns if the cache misses. In contrast to software prefetching, hardware prefetching does not require support from an optimizing compiler or a profiler. Furthermore, hardware prefetching can automatically adapt to the dynamic behavior of the application, such as varying data sets, or the hardware, such as systems with various cache sizes. Also, hardware prefetches are generated without the overhead of additional address-generation and prefetch instructions.

Stream prefetching is a well-known hardware prefetching technique that was proposed almost fifteen years ago [8]. Since its introduction stream prefetching has proved its usefulness by being introduced in different forms to virtually every modern high-performance microprocessors. As a consequence, researchers developing new prefetching mechanisms often compare their newly developed techniques to prefetching based on the idea of stream buffers.

Despite the fact that stream prefetchers are often used as a baseline to compare with new prefetching approaches, we could not find any kind of published material that analyzes the effectiveness of stream prefetchers on SPECcpu2000 programs in great detail. Nevertheless, in many cases new prefetching techniques are compared to stream prefetching, but the parameters of the stream prefetcher are not specified. In other cases the parameters are specified but the authors do not identify why they were chosen or whether these parame-

MSP'05, Chicago, USA.

ters are optimal for the benchmark suite and microprocessor under investigation.

This paper pursues two goals. The first goal is to provide a thorough analysis of stream prefetcher performance on all SPECcpu2000 programs and to identify the parameters that are the most critical for the stream prefetcher's effectiveness. We found the most crucial parameter to be the prefetch distance. Varying this parameter within a reasonable range can almost double the performance benefit. At the same time, the prefetch distance is probably the most ignored parameter in many prefetching studies. For instance, we analyzed the past 8 years of ISCA submissions and found that out of the four papers that use a stream prefetcher as a baseline for comparing with other prefetching techniques, only two specify the prefetch distance supported by the stream prefetcher.

The second goal of this paper is to provide an example of how non-optimal stream prefetcher parameters can distort evaluation results when stream prefetching is used as a baseline in a comparison with other prefetching schemes. To this end, we evaluate runahead execution [13] on several baselines with the same stream prefetcher but different prefetch distances. One may think that since runahead execution and stream prefetching target different types of load address patterns, the performance advantage of runahead execution over stream prefetching should stay about the same. However, our simulations show that changing the prefetch distance of the stream prefetcher from a suboptimal to the best value for the SPECcpu2000 benchmark suite reduces the average performance advantage of runahead execution from 13.7% to 7%. Note that we are not trying to say that runahead execution is a bad prefetching technique (we think it is great), but rather we are showing that using non-optimized stream prefetchers can results in an unfair comparison.

The rest of the paper is organized as follows. Section 2 provides an overview over the related work on stream prefetching. Section 3 describes the design of the stream prefetcher used in this study and the simulation methodology. Section 4 provides the sensitivity study of the stream prefetcher parameters and our case study of evaluating the performance benefit of runahead execution over stream prefetching with different parameters. Section 5 concludes the paper.

2. RELATED WORK

Hardware prefetching is an active area of research and several aggressive schemes have been proposed to mitigate the increasing performance impact of cache misses. In this work we use only the class of stream prefetchers, hence we limit our discussion to stride-based hardware stream prefetching mechanisms.

A stream prefetcher is usually located between the L1 and the L2 cache, monitoring the L1 cache miss requests that are delivered to the L2 cache. Stream prefetchers identify distinct streams within the sequence of the L1 cache misses, associate strides with each of the identified streams, and issue memory requests for the addresses further ahead in the stream. This kind of prefetching is usually effective for regular and structured workloads, including many high-performance and technical computing applications [8]. The simplest form of stride prefetching is next-sequential prefetching in which the prefetcher issues a request for line L+1 as soon as line L is referenced [3, 14]. Depending on the design complexity of the stream prefetchers, they can either detect streams with a unit stride or they can handle both unit and non-unit stride streams. A unit-stream prefetcher observes misses for sequential cache lines and prefetches additional sequential cache lines. More sophisticated stream prefetchers handle streams with non-unit strides, e.g., observing misses for cache lines L, L+4, and L+8, a non-unit stream prefetcher issues a prefetch for line L+12 [12]. A unit stream prefetcher will not be able to detect such a stream since it can only prefetch streams with the miss pattern L, L+1, L+2, etc. 2-delta stride prediction allows preventing single unrelated misses from causing the prefetcher to deviate from the correct stride [4].

In many cases the L1 cache miss addresses are composed of an interleaving of several streams. A typical case of multiple streams is the traversal of several arrays within a matrix-matrix multiplication loop. Stream prefetchers employ special mechanisms to decipher and disambiguate interleaving streams. If the memory hierarchy propagates the program counter (PC) of the instructions that cause cache misses beyond the L1 cache, the prefetcher can attribute misses to specific instructions and track streams on a per PC basis [1, 5, 6]. If PC information is not available, the stream prefetcher needs to identify distinct streams within the global memory access pattern. Minimum delta prediction and memory partitioning were proposed to handle this problem. Minimum delta prediction associates a miss with the stream or prior miss that is the closest [19]. Memory partitioning partitions the physical memory address space into regions and attributes all misses falling within a single region to a single stream [19]. The scheme used in this paper is based on the minimum delta approach.

The benefit from prefetching is largely determined by the timeliness of the issued prefetches. If prefetches are issued too late, they will hide only a fraction of the memory access latency. Stream prefetchers control the timeliness of prefetching by prefetching several stream elements ahead of the data consumption of the processor. This number of stream elements is called the prefetching distance and usually corresponds to the number of prefetches issued when a prefetcher learns a new stream. The optimal prefetch distance depends on several implementation-dependent factors, including the memory latency, the available bandwidth and the cache sizes. A very long prefetch distance may result in a lot of redundant prefetches, a waste of valuable memory bus bandwidth and pollution of the caches. If the prefetch distance is too short, prefetches issued by the stream prefetcher hide only a fraction of the memory latency and provide little benefit. To deal with this problem, it was proposed to temporarily delay prefetching or limit the prefetch ahead distance until it becomes more certain that a new stream has been identified [17]. Extending the training period has minimal performance impact but significantly reduces the negative effects of redundant prefetching [5]. Our experimental results are based on fixed prefetch distances.

A stream prefetcher issues prefetches only when encountering a missing load or additionally when detecting the use of a previously prefetched cache line by the processor. In the latter case, the cache notifies the prefetcher when the processor issues the first load to a cache line that was previously requested by the stream prefetcher. After that the prefetcher initiates additional prefetches. This is made pos-

sible by "tagging" the cache lines that are being delivered into the cache on behalf of the stream prefetcher. This way the cache controller can easily identify which cache lines were requested by the prefetcher. Tagging allows to completely avoid misses for a particular stream with the exception of the misses incurred while the behavior of the stream is learnt. Furthermore, since the prefetcher is informed as the prefetched data is consumed, additional prefetches can be issued in a more timely fashion [16]. The scheme described in this paper employs tagging.

Stream prefetchers either prefetch directly into the processor's caches or into special prefetch buffers that are accessed in parallel with the processor's caches. The original proposal of stream prefetchers by Jouppi assumed dedicated prefetch buffers, with each stream buffer holding the prefetched data for a single stream [8]. In the original implementation, only the head of each buffer was compared with the memory address of the requests, but later proposals added support to compare requests with all the elements in the stream buffers [5]. Prefetching into dedicated buffers ensures that incorrect prefetches do not pollute the caches [14]. However, the special storage for prefetched data does not only require additional area, but also complicates the design and verification associated with maintaining cache coherency between the stream buffers and the regular caches. The evaluations presented in this paper are based on directly prefetching data into the L1 cache.

3. EXPERIMENTAL SETUP

This section describes the organization and operation of the stream prefetcher used in this study as well as the architecture of the processor and the benchmark suite under investigation.

3.1 Stream Prefetcher Organization

In this paper, we investigate the performance of a stream prefetcher that is capable of detecting both ascending and descending streams with non-unit strides. To detect streams, the prefetcher monitors the addresses of all L1 cache misses. The data address of each cache miss is put into the miss history table of a fixed size. The entries are put into this table in FIFO order. Whenever a stream prefetcher observes an L1 cache miss, it searches the miss history table to find the miss address that is the closest, i.e., it computes the minimum difference between the current miss address and all previous miss addresses in the history table. After it determines this minimum delta, it checks the table again to see if this delta repeats twice. If such a match is found, the stream prefetcher engine allocates a new stream.

To allocate a new stream, the prefetcher needs to have an available stream entry in the stream table. The number of stream entries of a prefetcher determines how many active streams a prefetcher can handle at the same time. The stream prefetcher in this study uses an LRU replacement policy when allocating new entries. After the stream is allocated a stream table entry, the stream prefetcher starts issuing prefetches. In this paper, we refer to the number of prefetches issued after stream allocation as the prefetch distance. All prefetches are delivered directly into the cache and are tagged. If the processor consumes prefetched data, the stream prefetcher is notified of this fact. Upon receiving such a notification, the stream prefetcher looks up its stream table to see which stream entry the consumed address belongs to. If it finds a match, it increments the corresponding stream address by the stream's stride and issues one new prefetch in order to keep up with the data consumption of the processor.

We use a stream prefetcher that is capable of maintaining different prefetch distances for different levels of caches. This technique was derived from the stream prefetcher used in IBM's POWER 4 processor [18].

Our stream prefetcher can be characterized by three main parameters: the length of the miss history table, the number of stream entries, and the prefetch distance. In Section 4 we characterize how each of these parameters influences the performance of the stream prefetcher.

Table 1: Simulated processor parameters

Out-of-order execution	4-wide fetch/issue/commit, 128-entry ROB, 64-entry LSQ, 64 reservation stations, speculative scheduling, squash recovery
Functional units (latency)	4 Int ALUs (1), 2 FP ALUs (2), 2 Int Mul/Div units (3/20), 2 FP Mul/Div units (4/12), 2 memory ports, 1 cycle address generation for memory references
Branch prediction	Bimodal/gshare hybrid, 8k entries each, meta-predictor 8k entries, 16 RAS, 2k-entry 4-way BTB, 11 cycles minimum misprediction penalty, fetch stops at 1st taken branch
Memory system (latency)	64kB 2-way 64B line IL1 (2), 64kB 2-way 64B line DL1 (2), 1 MB 4-way 64B line L2 (20), main memory (400), memory bus operates at 1/4 CPU clock frequency

3.2 Simulation Methodology

We evaluate stream prefetching using an extended version of the SimpleScalar v4.0 simulator [11]. The baseline processor is a four-issue dynamic superscalar microprocessor similar to the Alpha 21264 (Table 1).

We used all integer and floating-point programs from the SPECcpu2000 benchmark suite with the exception of the four Fortran-90 programs. We excluded these programs because of the lack of a compiler. All programs are run with the SPEC-provided reference inputs. The C programs were compiled with Compaq's C compiler V6.3 025 using "-O3 -arch host -non_shared" plus feedback optimization. The C++ and Fortran 77 programs were compiled with g++/g77 V3.3 using "-O3 -static". We used the SimPoint toolset [15] to identify representative simulation points. Each benchmark is simulated for 500 million instructions after fast-forwarding past the number of instructions determined by SimPoint.

4. EXPERIMENTAL RESULTS

4.1 Prefetch Distance

In our first sensitivity study we vary the prefetching distance while keeping the number of stream buffers and the miss history parameters fixed. We chose to use 16 stream buffers and a history length of 16 misses because further increasing these parameters does not provide significant benefits as will be shown in the following subsections. Since our stream prefetcher can use different prefetch distances for the L1 and L2 caches, we perform two experiments to find the optimal distances for the two cache levels.

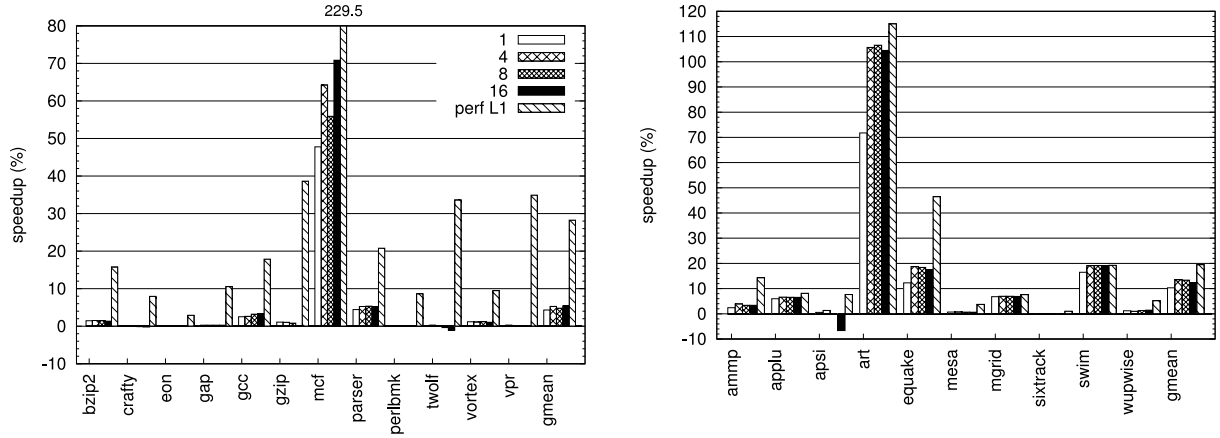

Figure 1: Sensitivity to the L1 cache prefetch distance

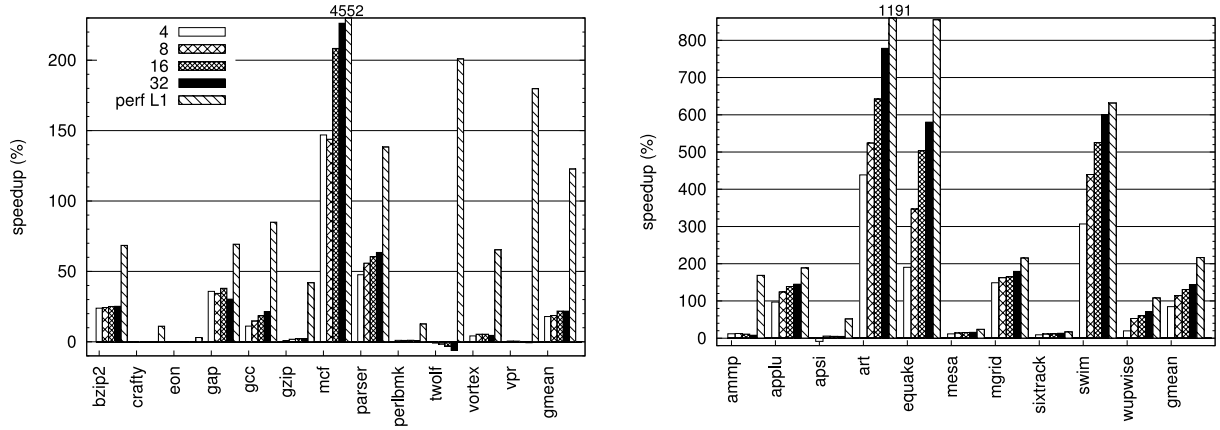

Figure 2: Sensitivity to the L2 cache prefetch distance

To investigate the impact of the prefetch distance on the L1 data cache performance, we simulate the baseline with a perfect L2 cache. Figure 1 shows speedup numbers for different prefetch distances. Even though we simulated all distances between 1 and 16, Figure 1 only shows selected parameters to highlight the general trend. The last bar for each benchmark represents the speedup with a perfect L1 cache, i.e., the case when the L1 cache always hits. This gives us an upper bound on the benefit attainable with prefetching.

On average, increasing the prefetch distance initially improves the performance. As the prefetch distance is increased from one to four, the average speedup for the integer programs increases from 4% to 5% and for the floating-point programs from 10% to 14%. After a certain point the negative effects of cache pollution cause slowdowns, and further increasing the prefetch distance does not provide any significant benefit for the integer benchmarks and results in a performance degradation for the floating-point benchmarks (the average speedup drops from 14% to 12%). In case of two benchmarks, *twolf* and *apsi*, higher prefetching distances hurt performance relative to the baseline without prefetching, which is mostly due to pollution.

Based on the results from Figure 1, it looks like the optimal prefetch distance for the L1 cache is four since this value yields the best performance with minimal pollution for our benchmark suite. Therefore, in all further experiments we fix the L1 prefetch distance to four for all programs.

Next, we investigate the performance benefit of varying the prefetching distance for the L2 cache from 4 to 32. The results are shown in Figure 2.

The best average speedup for the integer programs is achieved at a prefetching distance of 16. Further increasing the distance results in additional pollution, which is especially prominent in *gap* and *twolf*. However, the average speedup for the floating-point programs monotonically increases with the prefetch distance. A slight degradation is observed only for *ammp*. Even though larger prefetch distances result in significant pollution for both integer and floating-point programs, the prefetches hide so much of the memory access latency that they outweigh the negative effects of the pollution in almost all programs.

Another interesting fact to note is that the prefetching distance has a big impact on the performance benefit of the stream prefetcher. For example, changing the prefetch distance from 4 to 32 raises the speedup due to stream prefetching from 84% to 144% in case of the floating-point programs. Individual applications experience even more drastic improvements. Increasing the prefetch distance in *equake* improves the speedup from 190% to 565%, *swim* experiences an increase from 300% to 600% and shows a performance very close to that with a perfect cache. While the integer

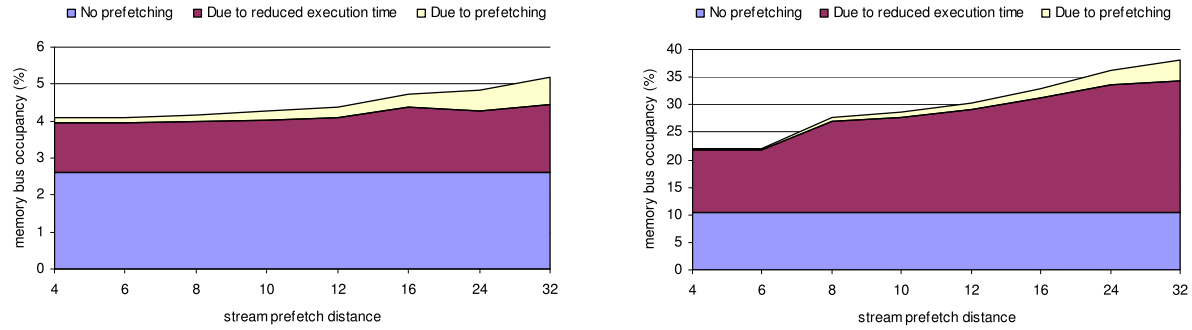

Figure 3: Memory bus utilization as a function of L2 cache prefetch distance

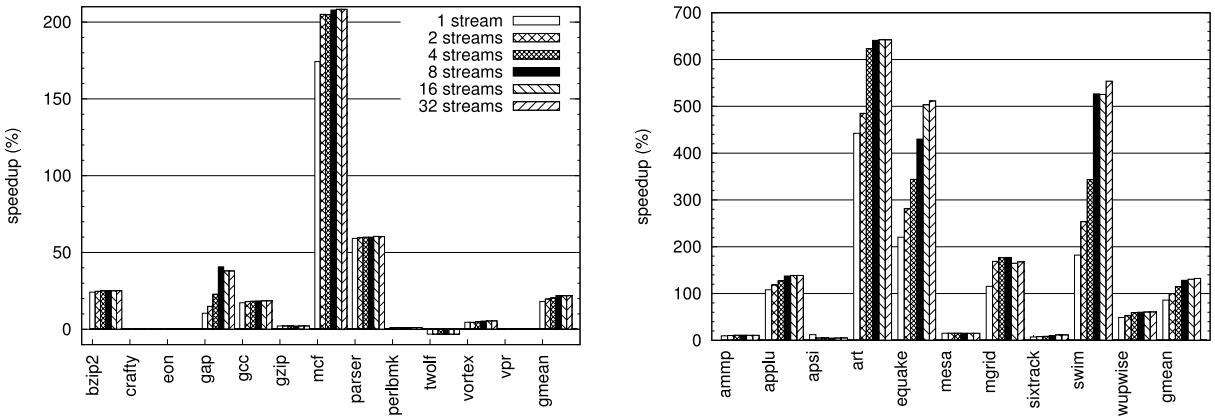

Figure 4: Sensitivity to the number of supported simultaneous streams

programs show less dramatic sensitivity to the prefetch distance, *gcc*, *mcf* and *parser* benefit greatly from an increased prefetch distance as well.

In general, the IPC improvements provided by our stream prefetcher correspond to the results in another recently published study involving stream prefetching [7]. While the absolute numbers for the performance improvements differ slightly, the trend for each program is similar.

Figure 3 shows the utilization of the main memory bus for various stream prefetch distances, averaged over the SPECint and SPECfp applications. The increase in bus utilization due to prefetching is divided into two parts: the increase caused by the reduced execution time and the increase caused by redundant prefetch traffic. Overall, the figure shows that the increase in bus utilization from 4.1% to 5.2% is quite tolerable for integer programs. On the other hand, bandwidth demands rise from 22% to 38.2% with increasing prefetch distances for the floating-point programs. Nevertheless, most of the increase comes from the faster execution. In the most aggressive configuration with a prefetch distance of 32, only 0.74% of the bus utilization is directly attributable to useless prefetching. In case of the floating-point programs, 3.77% of the total 38.2% bus utilization stems from redundant prefetching.

4.2 Number of Simultaneous Streams

In this subsection we investigate the sensitivity of the stream prefetcher to the number of streams that it can track simultaneously. We vary the number of streams supported by the prefetcher from 1 to 32 and measure the IPC improvement for each value. For all studies in this subsection, the prefetch distance for the L1 cache is set to 4, the prefetch distance for the L2 cache is 16, and the miss address history contains up to 16 last miss addresses.

The results are presented in Figure 4. For most integer and floating-point programs, increasing the number of concurrently supported streams beyond 16 provides little or no benefit. In general, some programs experience a relatively sharp increase in performance when the number of streams is increased from 1 to 8. *Equake* is the only program that shows a significant performance improvement when the number of supported streams is increased from 8 to 16.

Figure 4 shows that the SPECcpu2000 benchmark programs rarely exhibit more than 8 active streams at the same time. Therefore, to cover all possible cases it is important to provide a stream prefetcher with the capability to track at least 8 independent streams. Increasing the number of supported independent streams beyond 8 will not make a stream prefetcher more aggressive.

4.3 Miss History Length

The last parameter of the stream prefetcher that we investigate is the length of the miss history. To measure the sensitivity of the stream prefetcher to this parameter, we vary the history length from 2 to 64 and measure the IPC improvement for each value. For all studies in this subsection the prefetch distance for the L1 cache is set to 4, the prefetch distance for the L2 cache is 16, and the stream prefetcher can track up to 16 simultaneous streams.

The results are presented in Figure 5. To achieve the

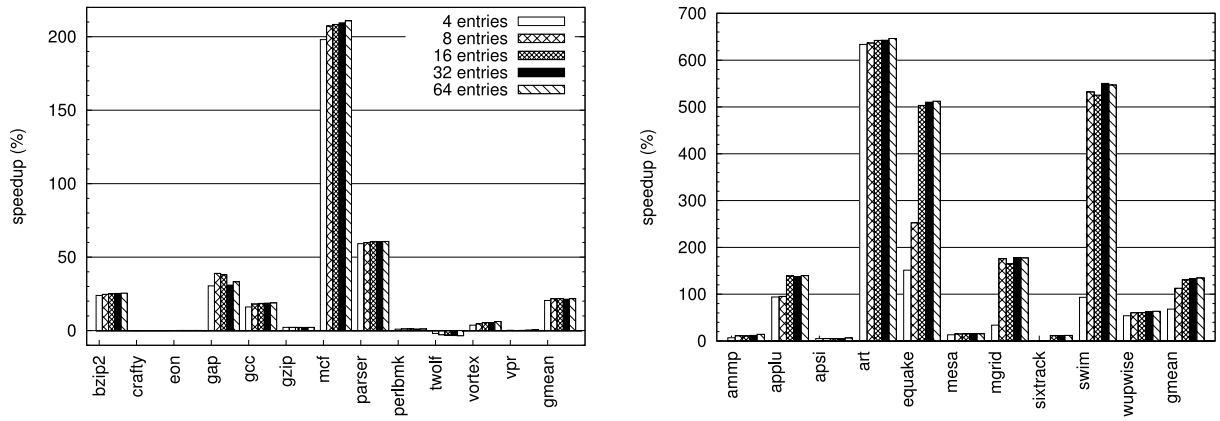

Figure 5: Sensitivity to the length of the miss history

maximum performance benefit, integer programs require at least 8 last miss addresses to be tracked by the stream prefetcher. However, a history of 4 entries already provides most of the benefit. The floating-point programs are more demanding and require a history length of at least 16 miss addresses. Increasing the history length from 4 to 16 boosts the speedup from 68.2% to 130.7% for the floating-point programs. Therefore, a history length of at least 16 is required for reaping most of the benefit provided by the stream prefetcher.

4.4 Case Study

The previous subsections showed that the stream prefetcher performance is highly dependent on the optimal configuration of its parameters. In this section we show that, to make a fair comparison with other prefetching techniques, it is important to choose the stream prefetcher with the best parameters.

As a case in point, we investigate the performance of prefetching based on runahead execution relative to stream prefetching. The concept of runahead execution was first proposed by Dundas and Mudge [2] for in-order processors and further extended by Mutlu et al. [13] to perform prefetching for out-of-order architectures. The runahead architecture "nullifies" and retires all memory operations that miss in the L2 cache and remain unresolved at the time they reach the ROB head. It starts by taking a checkpoint of the architectural state. Then it retires the missing load and the processor enters runahead mode. In this mode the instructions proceed largely normally except for two major differences. First, the instructions that depend on the result of the load that was nullified do not execute but are nullified as well. They commit an invalid value and retire as soon as they reach the head of the ROB. Second, store instructions executed in runahead mode do not overwrite data in memory. When the original nullified memory operation completes, the processor rolls back to the checkpoint and resumes normal execution. All register values produced in runahead mode are discarded.

We implemented a version of runahead execution to study its performance advantage over stream prefetching. Note that runahead execution is added on top of the stream prefetcher, i.e., we measure the performance benefit of using both a stream prefetcher and runahead execution over a baseline that uses only a stream prefetcher. We implemented the version of runahead execution that does not buffer intermediate memory state produced in runahead mode [13] and do not employ load-value prediction to assist runahead execution [10].

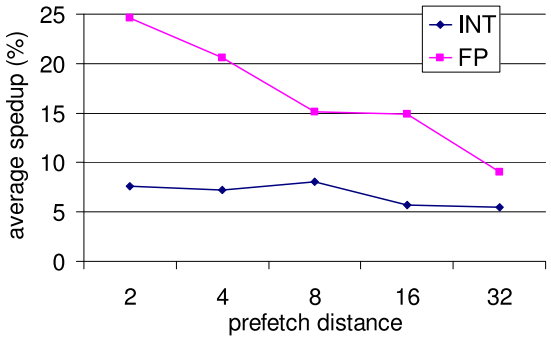

Figure 6: Average speedup of runahead execution over stream prefetcher with different prefetch distances

For this study, we fixed the number of stream prefetch buffers and the miss history to 16. We varied the prefetch distance of the stream prefetcher from 2 to 32 and measured the IPC for each configuration with and without runahead execution. Figure 6 presents the geometric-mean speedups of runahead execution over stream prefetching for integer and floating-point programs as the parameters of the baseline stream prefetcher are changed.

The data in Figure 6 demonstrate that the relative performance benefit of runahead execution is highly dependent on the parameters of the baseline stream prefetcher. As the prefetch distance of the stream prefetcher is increased, the average speedup of the integer benchmarks obtained by runahead execution drops from 7.6% to 5.4%. The floating-point programs exhibit a much more drastic drop from 24.6% to 9%. The average geometric-mean speedup for the entire SPECcpu2000 suite decreases from 13.7% to 7%, i.e., by a factor of two. While the performance increase obtained by runahead execution is quite good even with the most aggressive configuration of the stream prefetcher, this case study points out that the use of a non-optimally configured stream prefetcher can result in an unfair comparison of the stream prefetcher relative to other prefetching techniques.

The main reason for the decreasing performance gains with the increasing stream prefetch distance is the following. When a short prefetch distance is used, the stream prefetcher can hide only a fraction of the miss latency. Therefore, when the runahead processor issues loads that are already being prefetched by the stream prefetcher, they still take a long time. As a result, the processor enters runahead mode even though the requested data is already being prefetched. While in runahead mode, the processor keeps issuing memory requests that belong to the streams identified by the stream prefetcher during normal execution of the processor. The stream prefetcher observes that the prefetched data is being used and advances the prefetching front, which results in an increased prefetching distance while in runahead mode. Therefore, much of the benefit comes from the fact that runahead execution indirectly increases the prefetch distance of the stream prefetcher.

As we increase the prefetch distance, the stream prefetches hide more and more latency and the processor enters runahead mode less often and for shorter periods of time. Therefore, the benefit of runahead execution due to the advancing of the prefetching front while in runahead mode decreases. Since floating-point programs are more sensitive to the prefetch distance, the performance drop for them is quite significant. Figure 2 demonstrates that integer programs are less sensitive to the prefetch distance, which is why the benefit of runahead execution decreases more gradually.

5. CONCLUSION

This paper provides a detailed analysis of how the parameters of hardware stream prefetchers affect the memory system performance in SPECcpu2000 programs. We experiment with different parameters of the stream prefetcher and demonstrate that the same prefetching mechanism can deliver drastically different performance benefits depending on the configuration used. For example, increasing the prefetch distance from 4 to 32 almost doubles the performance benefit delivered by the stream prefetcher in floating-point programs. Therefore, we argue that parameter optimizations are imperative when comparing a stream prefetcher with other prefetching techniques. To emphasize the importance of parameter optimization, we investigate the performance of hardware prefetching based on runahead execution relative to stream prefetching and show that choosing a non-optimal stream prefetcher as a baseline significantly distorts the results of such a comparison. When a stream prefetcher with sub-optimal parameters is used, runahead execution provides 13% speedup over stream prefetching. When the parameters of the stream prefetcher are adjusted to achieve the maximum speedup for the given benchmark suite and processor architecture, the average speedup provided by runahead execution drops to 7%.

6. REFERENCES

[1] F. Dahlgren, M. Dubois, and P. Stenstrm. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 6(7):733–746, 1995.

[2] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th international conference on Supercomputing*, pages 68–75, 1997.

[3] R. J. Eickemeyer and S. Vassiliadis. A load instruction unit for pipelined processors. *IBM Journal of Research and Development"*, 37(4):547–564, July 1993.

[4] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 133–143, 1997.

[5] K. I. Farkas, N. P. Jouppi, and P. Chow. How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors? In *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, page 78, 1995.

[6] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *Proceedings of the 25th annual international symposium on Microarchitecture*, pages 102–110, 1992.

[7] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham. Effective stream-based and execution-based data prefetching. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 1–11, 2004.

[8] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th annual international symposium on Computer Architecture*, pages 364–373, 1990.

[9] S. Kalogeropulos, M. Rajagopalan, V. Rao, Y. Song, and P. Tirumalai. Processor aware anticipatory prefetching in loops. In *Proceedings of the tenth international symposium on High Performance Computer Architecture*, pages 106–117, 2004.

[10] N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martinez. Checkpointed early load retirement. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 16–27, 2005.

[11] E. Larson, S. Chatterjee, and T. Austin. Mase: a novel infrastructure for detailed microarchitectural modeling. In *Proceedings of the The Second International Symposium on Performance Analysis of Systems and Software*, pages 1–9, 2001.

[12] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 62–73, 1992.

[13] 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 The Ninth International Symposium on High-Performance Computer Architecture*, page 129, 2003.

[14] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st annual international symposium on Computer architecture*, pages 24–33, 1994.

[15] T. Sherwood, E. Perelman, G. Hamerly, and

B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, 2002.

[16] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 42–53, 2000.

[17] V. Srinivasan, E. S. Davidson, and G. S. Tyson. A prefetch taxonomy. *IEEE Trans. Comput.*, 53(2):126–140, 2004.

[18] J. M. Tendler, J. S. Dodson, J. S. F. Jr., H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[19] C. Zhang and S. A. McKee. Hardware-only stream prefetching and dynamic access ordering. In *Proceedings of the 14th international conference on Supercomputing*, pages 167–175, 2000.