Two-level Data Prefetching

Fei Gao[†], Hanyu Cui[§], Suleyman Sair[§]
[†] MathWorks, Inc.
[§] Department of Electrical and Computer Engineering, North Carolina State University
fei.gao@mathworks.com, {hcui, ssair}@ece.ncsu.edu

Abstract

Data prefetching has been shown to be an effective tool in hiding part of the latency associated with cache misses in modern processors. Traditionally, data prefetchers fetch data into a small prefetch buffer near the L1 for low latency, or the L2 cache for greater coverage and less cache pollution. However, with the L1–L2 cache speed gap growing, significant performance gains can be obtained if the data prefetcher can operate as aggressively as an L2-level prefetcher but with the fast hit times of an L1-level prefetcher. In this paper, we propose a prefetching framework where an L1-level prefetcher and an L2-level prefetcher work cooperatively to reduce the average access time more than either one alone can. We evaluate several design alternatives suited to perform synergistically under different workloads. From the insight we gather from this analysis, we propose a confidence-based adaptive prefetcher that can improve prefetch efficiency significantly with judicious use of available bus bandwidth. Our results show that for certain prefetcher combinations, two-level prefetching can achieve the cumulative speedup attained from either prefetcher alone. Furthermore, when compared to other two-level prefetching models, the adaptive design provides similar speedups with appreciably less bus traffic.

1 Introduction and Motivation

Data prefetching is one of the commonly used techniques in mitigating the negative effects of long cache miss latencies on program execution time. Whether driven by compiler inserted instructions, or issued at runtime by a hardware prefetcher, prefetching aims to hide the latency of a would be cache miss by initiating the data access sooner. This speculative access brings the prefetched data into the caches and/or a dedicated prefetch buffer.

Usually, prefetching data directly into the L1 data cache is avoided to prevent cache pollution. Instead, a small buffer that has a comparable access time to the L1 cache, stores the prefetched data until it is used or replaced. Throughout this paper, we use the term L1 prefetcher to indicate this kind of prefetching. The alternative is prefetching into the L2 cache – referred to as an L2 prefetcher in the paper. While this option does not result in significant cache pollution in most cases, it doesn't hide as much latency as a prefetch buffer. Another striking difference between these two types of prefetching schemes is how aggressive they can get in issuing prefetches. Since the prefetch buffer is relatively small when compared to the caches, an L1 prefetcher needs to be very selective in which addresses to fetch. For this reason, these prefetchers tend to have a high prefetching accuracy but low prefetch coverage. On the other hand, a scheme prefetching into the L2 is less restricted by space, so it usually generates a lot of prefetches trying to cover as much ground as possible. This leads to lower accuracy but higher coverage. In short, each prefetcher optimizes a different part of the memory hierarchy by design.

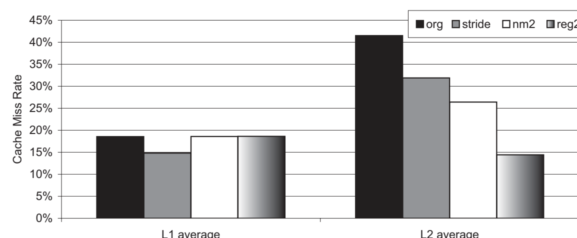

Figure 1: Average L1 and L2 cache miss rates obtained with our baseline architecture with no prefetching, a stride L1 prefetcher, a hybrid next-line/markov L2 prefetcher and a scheduled region prefetching L2 prefetcher. The cache configuration is shown in Table 1.

Figure 1 illustrates the average L1 and L2 cache miss rates with and without prefetching for the set of benchmarks we analyze in this paper. The L1 prefetcher is a stream buffer style stride prefetcher [5]. We also show results for two L2 prefetchers, a moderately aggressive hybrid next-line/markov prefetcher [4] (labeled nm2) and an ambitious L2 prefetcher in the scheduled region prefetcher [6] (labeled reg2). The benchmark descriptions as well as prefetcher specifics are presented in Section 3. The results show that the L1 prefetcher focuses on reducing the L1 miss rate and is not as effective in reducing the L2 miss rate. Meanwhile, the L2 prefetchers exclusively work on reducing the number of L2 misses. When used together however, these two prefetchers can have a profound impact on performance as

1-4244-1258-7/07/$25.00 ©2007 IEEE

the gap in L1-L2 access times is ever widening with the inclusion of smaller L1 caches but bigger L2 caches with each processor generation. In addition, bus bandwidth needs to be judiciously utilized in the face of CMP designs.

This paper argues for an integrated approach in data prefetching that aims to ameliorate both L1 and L2 hit rates at the same time. Along with an accurate L1 prefetcher, we propose to simultaneously use an L2 prefetcher in a cooperative manner. In the ideal case, the L2 prefetcher reduces the turnaround time for L1 prefetches as well as reducing the average latency for demand misses. With a reduced prefetch turnaround time, hits in the L1 prefetcher are more likely to hide the full miss latency or a larger chunk of it. In contrast, in the worst case, the L1 and L2 prefetchers would slow one another by creating a lot of traffic on the L2-Memory bus. We put forward several two-level prefetching designs that promote cooperation rather than competition among the prefetchers.

The rest of the paper is organized as follows. In Section 2, prior hardware prefetching models are discussed. Simulation methodology and benchmark descriptions can be found in Section 3. Section 4 describes our prefetching models while Section 5 presents an evaluation of their merits. Finally, our conclusions are summarized in Section 6.

2 Related Work

In the Markov prefetcher described by Joseph and Grunwald [4] each missing address would index into a Markov prediction table to provide the set of cache addresses that have followed this address before. After these addresses are prefetched, the prefetcher stays idle until the next cache miss. Scheduled region prefetching [6] (SRP) is an aggressive demand-based prefetching technique that fetches the data surrounding an L2 cache miss. This surrounding region is typically the same size as a memory page. Subsequently, Wang et al. build on the SRP framework to implement a cooperative software-hardware prefetcher [9]. In this scheme, the compiler inserts prefetch hints for load instructions and the region prefetcher would issue the prefetches in hardware.

Stream buffers follow multiple data streams, prefetching them in parallel [5]. They prefetch consecutive cache blocks, starting with the one that missed in the L1 cache. Farkas et al. [2] extend the stream buffers to use a PC-based stride predictor to provide a stride that is used to generate the next prefetch address instead of prefetching consecutive blocks. Farkas and Jouppi [3] further enhance the stream buffer design by enforcing the streams being followed by multiple stream buffers to be non-overlapping. This prevents duplication and saves bus bandwidth. In a further enhancement, Sherwood et al. propose a new form of stream buffer called the Predictor-Directed Stream Buffer [8]. Instead of associating a fixed stride with each buffer, they propose a framework where any address predictor can generate the next address to prefetch. Then, that prefetch address can be recursively used to generate the next prefetch address. In particular, they present a hybrid stride filtered markov predictor to direct stream buffer prefetching and find that it can accurately prefetch both striding and pointer-based workloads.

Parameter	Value
Fetch/Decode/Retire width	8 instructions
RUU size	128
Load/store queue size	64
Function units	8 intALU, 2 int mul/div
I-cache L1	ideal
Brand Pred.	2K entry hybrid bimodal and gshare
D-cache L1	32KB, 2-way set assoc., 32 byte lines, 3 Cycle hit latency, 16 MSHRs
L2 cache	1MB, 4-way set assoc., 64 byte lines, 18 Cycle hit latency, 300 Cycle miss latency
L1/L2 data bus bandwidth	8B/cycle
L2/MEM data bus bandwidth	4B/cycle

Table 1: Architectural configuration.

In our analysis, we evaluate a decoupled L1 prefetcher guided by a PC-based stride predictor. The prefetches are stored in a unified stream buffer. For the L2 prefetcher, we experiment with a Markov prefetcher and a scheduled region prefetcher to represent different levels of prefetcher aggressiveness.

3 Methodology

The simulator used in this study was derived from the SimpleScalar/Alpha 3.0 tool set [1], a suite of functional and timing simulation tools for the Alpha AXP ISA. We rewrote the SimpleScalar memory system as an event-driven model with detailed simulation of memory and bus operations. Table 1 presents the configuration parameters for the baseline system. Since instruction footprints of SPEC benchmarks are fairly small and The effect of instructions on L2 and memory traffic is minimal, we simulated with an idealized instruction cache to reduce simulation time.

This work studies 8 memory-intensive benchmarks. `Mcf`, `parser`, `applu`, `mgrid` and `equake` are from SPEC2000 suite, while `swim` is from SPEC95. We also used `deltablue`, an incremental constraint hierarchy solver and `dot`, which is a tool for automatically making hierarchical layouts of directed graphs. The programs were compiled on a DEC Alpha AXP-21164 processor using the DEC C and C++ compilers under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`). Each program was simulated for 100 Million committed instructions after skipping the initialization part (except for `deltablue`) as determined by the SimPoint toolkit [7]. The processor state is warmed up before detailed simulation starts. The benchmark details are presented in Table 2.

To perform our evaluation, we chose three prefetchers

Benchmarks		Input	IPC	%DL1 MR	%L2 MR	FFwd
SPEC95	swim	ref.in	1.65	7.2	46.5	0.6B
SPEC2K	mcf	inp.in	0.06	75.3	90.3	5.3B
	parser	ref.in	1.27	5	16.3	21.7B
	applu	applu.in	1.23	10.7	48.6	217.9B
	mgrid	mgrid.in	1.62	6.2	28.2	3.29B
	equake	inp.in	0.63	16.8	38	81.2B
MISC	deltablue	long	1.29	17.1	2.7	0
	dot	in.dot	0.17	29.3	80.5	5.1B

Table 2: The benchmarks studied in this paper. We also list the the baseline IPC, L1 data and L2 cache miss rates and the number of instructions skipped before starting performance simulation.

at different points of the prefetching intensity spectrum. For accurate L1 prefetching, we use a 128 entry stride prefetcher. The address predictor is a PC-indexed, 256-entry, 4-way two-delta stride predictor trained with the L1 miss stream. For moderate L2 prefetching, we utilized a hybrid nextline/markov prefetcher. This prefetcher has a large 32K entry markov predictor that holds the next 4 possible transitions for each entry. If the markov table misses, we generate a nextline prefetch for this address. To implement an aggressive L2 prefetcher, we employ the scheduled region prefetching scheme, which queues up cache blocks in the surrounding 2KB region of the missing address and prefetches them when the L2–Memory bus is free. For each prefetcher at a certain level of the memory hierarchy (e.g. L1 prefetcher), we probe the prefetch queue and the cache at that level before each prefetch is issued to the next level.

4 Two-level Prefetcher Design

In our taxonomy of the two-level prefetching hierarchy, there are two parameters, each with two possible values. We will describe these models and evaluate their merits in this section.

The main goal of a two-level prefetching approach is to reap the benefits of both prefetchers at the same time without diminishing their effectiveness. In this best case scenario, the L2 prefetcher forges ahead of the L1 prefetch stream, reducing the turnaround time for L1 prefetches as well as mitigating the average latency for L2 demand misses. When the L1 prefetch data return to the prefetch buffer in a shorter amount of time, they have a bigger chance of hiding the complete memory latency.

Figure 2 depicts a simple memory hierarchy augmented with an L1 and an L2 prefetcher, labeled L1-p and L2-p respectively. The demand accesses are shown on the left hand side with white arrows. L2 prefetcher requests, fetching data from the memory to the L2 cache, are displayed with the black arrows. The L1 prefetcher requests are the gray arrows. If we do not allow the L1 prefetcher access to the memory, the light gray path is blocked. Also pictured are the training streams for the prefetchers. The L1 prefetcher is trained with the demand L1 miss stream. For the L2 prefetcher, in addition to the demand L2 miss stream,

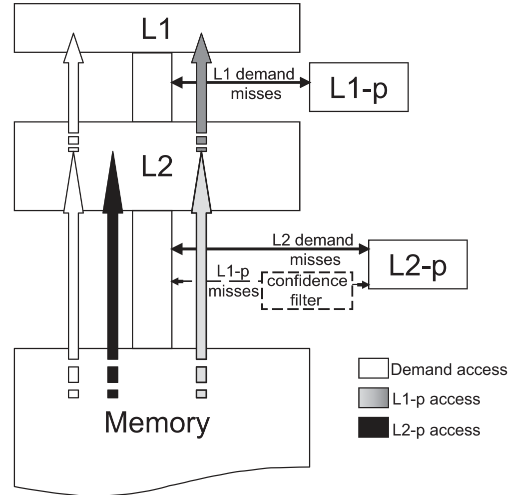

Figure 2: The flow of data and prefetch meta-data. Depending on the two-level prefetching model, some the paths shown in light gray or dashed lines are restricted.

we could train the address predictor with the L1 prefetch misses as shown with the dashed line. And finally, as we will detail in Section 4.2, instead of training with all L1 prefetcher requests, we can filter through only the confident ones to assure maximum benefit at the lowest possible bus bandwidth consumption.

The models we investigate mainly differ along two different axes: 1) L2–Memory bus access and, 2) L2 prefetcher training stream. An L1 prefetcher brings data from the memory to the L2 cache and from the L2 cache to the prefetch buffer. The L2 prefetcher fetches data from memory into the L2 cache only. We call this operation **shared** mode since the prefetchers share the L2–Memory bus. In contrast, when the prefetchers operate in **exclusive** mode, the L1 prefetcher is only allowed to fetch data from the L2 cache to the prefetch buffer while the L2 prefetcher is responsible for fetching data from the memory to the L2. In other words, each prefetcher can only access the bus that brings data into the level it resides. These two modes of operation allow us to control the amount of bus traffic that is induced by the prefetchers.

Normally, the address predictors that determine the prefetch stream are trained with demand miss addresses. We call this **demand-only** prefetching. Alternatively, we can add L1 prefetch misses to the L2 prefetcher training stream in addition to the demand misses. This allows the L2 prefetcher to anticipate and prefetch L1 prefetch requests, reducing their transfer time into the prefetch buffer. This is referred to as the **cooperative** mode in the paper. Note that all four pairwise combinations along these two axes can be useful depending on application characteristics.

4.1 Two-level Prefetching Models

Now that we have defined the different design points we consider for a two-level prefetcher, we can dissect the four possible combinations .

Serial: In a *serial* model, L1/L2 prefetchers work in exclusive and demand-only mode. It does not permit L2–Memory activity stemming from the L1 prefetcher. In addition to reducing bus traffic, this also decreases the likelihood of conflicts and pollution in the L2 cache. However, this limits how far ahead of the demand miss stream the L1 prefetcher can get.

Parallel: In this model, L1/L2 prefetchers work in shared and demand-only mode. Both prefetchers operate oblivious to each other's existence. Because it has no restrictions on bus accesses, it generates a lot of traffic on the L2–Memory bus. On the positive side, if the L2 prefetcher can get ahead of the L1 prefetch stream, it could reduce the turnaround time of L1 prefetches.

Delegation: One shortcoming of serial mode is that the L1 prefetcher is rendered ineffectual when the L2 prefetcher does not fetch along an overlapping stream. In *delegation* mode, we eliminate this problem by training the L2 prefetcher with the L1 prefetch misses (cooperative mode). In a sense, the L2 prefetcher is delegated to bear the burden of handling L1 prefetch misses in the L2 cache. Note that this is different from the parallel model where we would only fetch the missing block from the memory to the L2 cache on an L1 prefetch miss. In delegation mode, depending on the particular L2 prefetcher, we could be fetching multiple blocks following/surrounding the miss. Most importantly, the L2 prefetcher can get ahead of the L1 prefetch stream and warm up the L2 cache.

Drafting: The *drafting* model is similar to delegation. The only difference is that the prefetchers work in shared mode instead of exclusive mode. This ensures that in addition to the future set of L1 prefetch requests, the current request is also serviced by the slower levels of the memory hierarchy. The goal here is to enable the L2 prefetch stream get ahead of the L1 prefetcher, and maximize the effectiveness of the L1 prefetch stream. When compared to delegation, drafting amounts to a more significant reduction in the average latency of L1 prefetch requests. Note that this could also lead to an appreciable number of L2 cache conflicts due to extra data being brought into the cache by the L2 prefetcher on behalf of the L1 prefetcher.

4.2 Adaptive Cooperative Prefetching

Cooperative prefetching can be taxing on the L2–Memory bus, especially for programs with a high L2 miss rate. To reduce the contention for L2 cache entries and the memory bus, we propose an adaptive filtering scheme.

In general, the L1 prefetches that will hide the most latency are those that belong to an accurate prediction stream. Accordingly, improving the timeliness of these L1 prefetches should bring the most benefit. In our filtering scheme, we rely on the address predictor for the L1 prefetcher to indicate which prefetch streams are confident. Specifically, we increment a 2-bit saturating counter in the stride address predictor each time a prefetch is used by the processor. A stream is deemed confident if its confidence counter is above a certain threshold (2 in our case). Thereafter, only L2 misses coming from these streams are allowed to trigger L2 prefetches.

5 Evaluation

We present the quantitative analysis of the proposed two-level prefetching models in this section. We use several two-level prefetching architectures. The L1 prefetcher in these architectures is a unified stream buffer style stride prefetcher. For the L2 prefetcher, either a relatively moderate nextline/markov prefetcher (NM), or toward the aggressive end, a scheduled region prefetcher (Reg) is used. For completeness, we also present results with another Stride prefetcher at the L2 level in Section 5.2. When presenting relative results, we normalize them to those of our baseline architecture with no prefetching.

5.1 Exclusive L2-Memory Bus Access Mode

When prefetcher access to the L2–Memory bus is exclusive, the two-level prefetcher can be in either serial or delegation mode. Figure 3 shows the performance impact of the exclusive L2 bus modes. In the top graph, the L1 prefetcher alone gets higher performance than the NM L2 prefetcher alone. None of the two-level prefetcher modes can outperform the L1 prefetcher. This is because the L1 prefetching stream is very accurate. Terminating the requests of the L1 prefetcher that miss in the L2 cache results in a great loss of opportunity to prefetch useful data. And since the performance gain from the L2 prefetcher does not make up for the loss of L1 prefetcher, a standalone L1 stride prefetcher works better than the two-level prefetching models.

The bottom graph in Figure 3 uses a Reg L2 prefetcher. Because the Reg L2 prefetcher is more aggressive, it covers many more misses than the NM prefetcher. As a result, we can see that the prefetcher gains have reversed in that the L2 prefetcher enjoys a higher speedup than the L1 prefetcher. However, similar to the NM L2 prefetcher, we still don't see noticeable synergy in the two-level prefetching modes. But, unlike the previous graph, the two level prefetching modes maintain the effectiveness of the better prefetcher (i.e. the Reg L2 prefetcher) since these models do not restrict the L2 prefetcher.

From these two level prefetcher combinations, we can conclude that exclusive L2 bus access is not a favorable two-level prefetching model for these benchmarks because it isolates and restricts the prefetchers. However, it can be the only viable model in certain cases. One such example is a CMP with a shared L2 cache where the L2 capacity and L2–Memory bus bandwidth would already be under heavy strain from applications running on multiple cores.

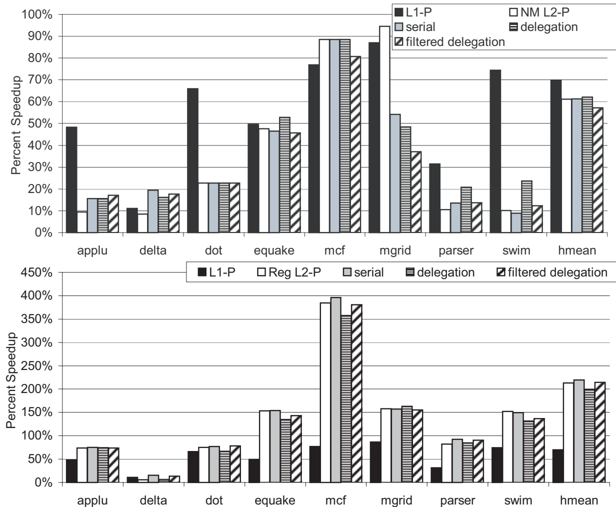

Figure 3: Exclusive Mode: Percent speedup results for the stride prefetcher, NM L2 prefetcher (top), Reg L2 prefetcher (bottom), and two-level prefetching configurations using serial, delegation, and filtered delegation. The results are with respect to the baseline architecture with no prefetching.

5.2 Shared L2–Memory Bus Access Mode

We will now analyze the performance of the shared bus access mode. When the L1 prefetcher is also allowed to access the L2–Memory bus, the two level prefetcher can be in either parallel or drafting mode. There are three sources of L2 bus requests in shared L2 bus mode - demand requests, the L1 prefetcher and the L2 prefetcher. This results in more bus traffic compared to exclusive L2 bus access mode. The increasing bus traffic has significant performance impact depending on how useful these extra memory requests are. Figure 4 shows the speedup for three different L2-prefetchers: (1) NM (top chart), (2) Stride(middle chart), and (3) Reg (bottom chart). In addition to the two main L2-prefetchers we evaluate, we look at a Stride L2 prefetcher to see how the significant overlap between the L1 and the L2 prefetcher affect performance.

For the NM prefetcher, we see 92%, 42% and 55% performance gains for parallel, drafting and adaptive modes respectively compared to the standalone L1 prefetcher. Although drafting gets lower average performance than parallel, the loss is mostly from `mcf`. `mcf` is a very bus traffic intensive benchmark, as discussed later in Figure 5 and the added pressure from drafting degrades timeliness of prefetches. It is interesting to note that the average speedup from the parallel mode is greater than the sum of the separate speedups from the L1 and the L2 prefetchers.

When both the L1 and the L2 level prefetchers are based on Stride-based address predictors, we see almost no benefit from two-level prefetching. This is an intuitive result because the two prefetchers are prefetching along the same streams without any lag between them obviating the need for a second prefetcher.

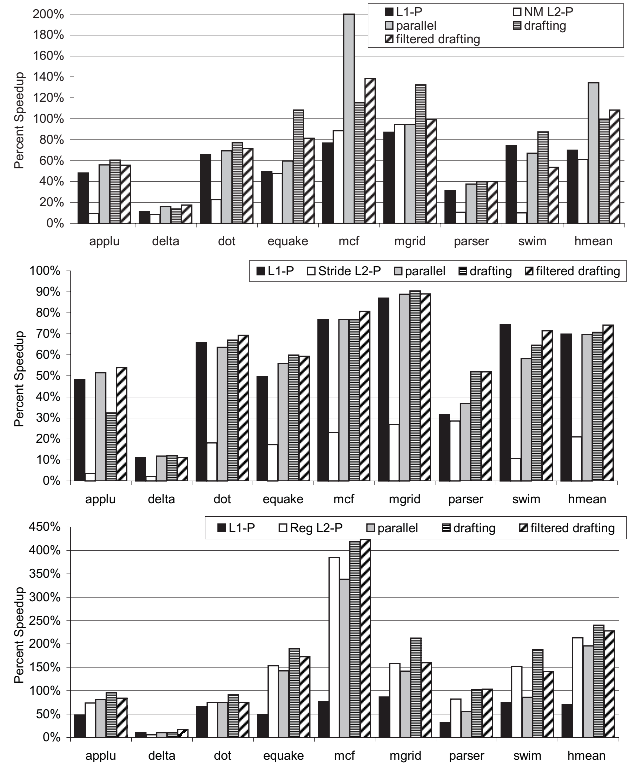

Figure 4: Shared Mode: Percent speedup results for the stride prefetcher, NM L2 prefetcher (top), Stride L2 prefetcher (middle), Reg L2 prefetcher (bottom), and two-level prefetching configurations using parallel, drafting, and filtered drafting. The results are with respect to the baseline architecture with no prefetching.

For the Reg prefetcher, we can observe that it obtains speedups of -8%, 13% and 7% for parallel, drafting and filtered drafting respectively when compared to the standalone L2 prefetcher. Parallel mode gets worse performance than Reg L2 alone. This is because Reg is a very aggressive prefetcher, which creates significant L2 bus traffic. When we add the L1 prefetcher on the top of the Reg L2 prefetcher, the performance loss is more than the gain from this extra traffic. However in drafting and filtered drafting, the Reg L2 prefetcher helps the L1 prefetcher by issuing requests for L1 prefetcher misses in the L2 cache. Because the L1 and the L2 prefetchers are of different types, the Reg L2 prefetching can only cover the stride-based L1 prefetch stream when it is trained with L1 prefetcher misses. This reduces the turnaround time of L1 prefetching requests to improve performance.

When compared to the exclusive mode results discussed in Section 5.1, the shared modes outperform the exclusive access modes for most of the benchmarks. Thus, we focus on the shared mode results in the reminder of the paper.

5.3 Other Prefetcher Efficiency Metrics

Having observed the performance potential of two-level prefetching, we now turn our attention to the metrics that provide an insight into why we attain these speedups.

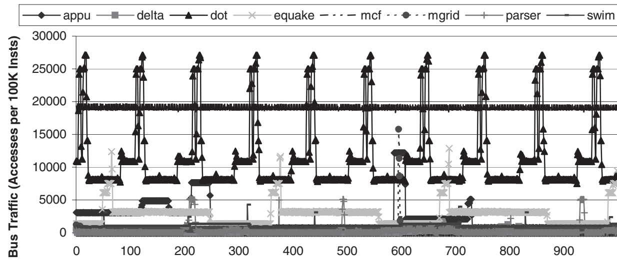

Figure 5: L2–Memory bus traffic density behavior plotted over a 100K-instruction sampling period. The y-axis is the number of requests in 100K instructions.

Figure 5 presents the bus traffic behavior during program execution. This figure plots the percentage of time the L2–Memory bus was busy during program execution. The bus traffic is sampled at a fixed sampling period (100K committed instructions in this case). We can see that `mcf` has a high bus utilization throughout the whole execution. `dot` also exhibits bursts of high bus activity where a big cluster of misses overwhelm the bus. This is where filtering the streams targeted by the L2 prefetcher on behalf of the L1 prefetcher is useful. In most cases, filtered drafting attains similar speedups to drafting while utilizing the bus 10% less frequently on average.

Meanwhile, Figure 6 shows the breakdown of average load latency. There are eight structures that can service a memory request: L1 cache, L1 prefetch buffer, L1 MSHRs, L1 prefetcher MSHRs, L2 hit, L2 MSHRs, L2 prefetcher MSHRs, and finally the main memory. The total average latency corresponds to the speedup obtained from each scheme. Note how the clustered misses in `dot` and `mcf` result in a very high L1 MSHR component. Two-level prefetching models get a lower average latency in general. Going from parallel to drafting reduces the average access latency by prefetching L1 prefetch requests that would have been L2 misses. Filtered drafting performs close to drafting but with less bus utilization.

The motivation for two-level prefetching is to combine the benefit of the standalone L1 and L2 prefetchers. The L1 prefetcher can increase the percentage of loads that have an L1-level access time (i.e. L1 cache + L1 prefetch buffer). Meanwhile, the L2 prefetcher focuses mostly on reducing the number of L2 misses by prefetching data into L2 cache. In addition, it reduces the turnaround time for L1 prefetch requests by prefetching on behalf of the L1 prefetcher, which in turn results in fewer L2 cache demand accesses because data is found in the L1 prefetch buffer. Figure 7 shows the L1 level hit rate changes for different one-level and two-level prefetching schemes with Reg L2 prefetching. We can see that two-level prefetchers get similar or higher L1 hit rates than L1 prefetcher alone. For `mcf`, the L1 hit rate is significantly higher than the original L1 hit rate of L1 prefetcher alone because of the existence of an L2 prefetcher.

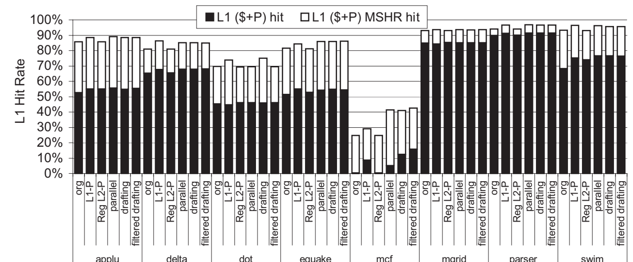

Figure 7: Number of L1 hits for the different configurations examined in the paper.

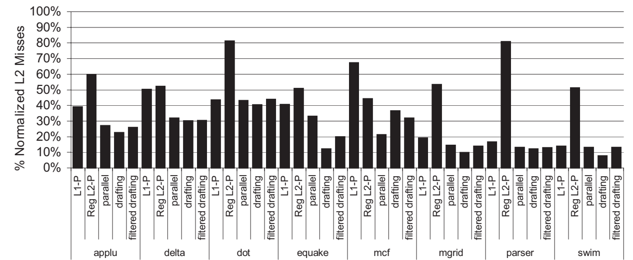

Figure 8: Normalized number of L2 miss results for different cooperative two-level prefetching schemes using the Reg L2 prefetcher. Results are normalized to those of the baseline architecture with no prefetching.

We depict the supporting data for the other component of two-level prefetching synergy in Figure 8. Figure 8 depicts the L2 miss coverage, which is normalized to the original L2 misses without a prefetcher. Again, we see that two-level prefetching models have significantly fewer L2 misses than L2 prefetcher alone. The L1 and L2 cache performance graphs verify that two-level prefetcher inherits the benefit of both standalone prefetchers.

5.4 Comparison of parallel, drafting and adaptive modes

The key difference between parallel, drafting and filtered modes is the relationship between the L1 prefetcher and the L2 prefetcher (i.e. how the L2 prefetcher helps the L1 prefetcher). Figure 9 shows the breakdown of the L2 accesses stemming from the L1 prefetcher. The data requested by the L1 prefetcher can be brought into the L2 cache from three sources - demand requests, L1 prefetcher requests and L2 prefetcher requests. Since the L2 prefetcher prefetches data for L1 prefetcher in drafting and filtered drafting modes, we can see the increasing portion of misses serviced by the L2 prefetcher. And because filtered drafting is confidence based where only confident L1 prefetch streams train the L2 prefetcher, it has lower L2 prefetcher service coverage than the drafting scheme.

6 Conclusions

The continuing trend of smaller feature sizes will make the gap between logic and memory speeds growing continually. New processor versions feature smaller L1 caches for higher clock frequency. Yet the L2 cache size grows with each new

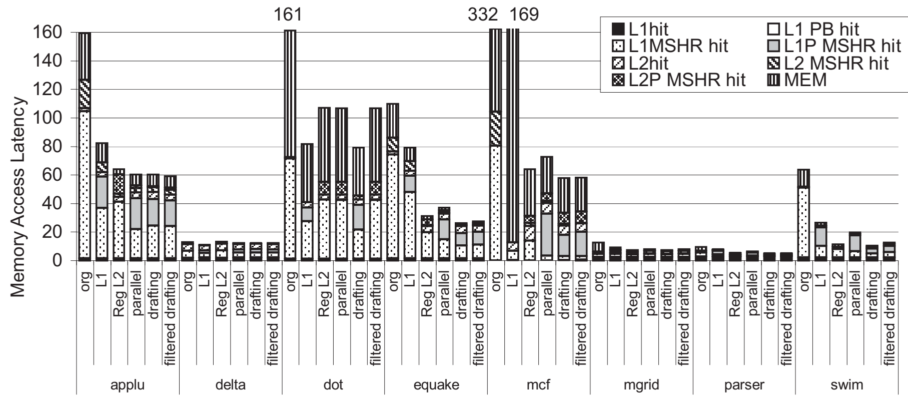

Figure 6: Average memory access latency broken down into parts corresponding to the memory hierarchy structure that supplied the data.

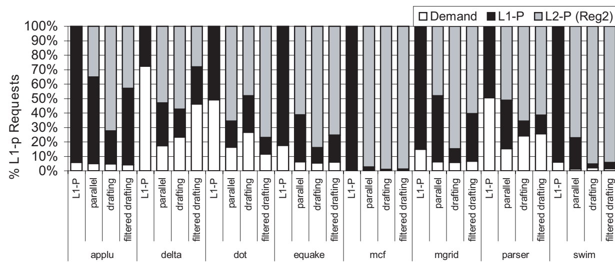

Figure 9: The breakdown of L1-prefetcher requests that were serviced by the L2 cache. Each component corresponds to the structure that brought the requested cache block into the L2 cache.

generation. While out-of-order execution helps hide short latencies, other techniques that mask the growing L1-L2 access latency gap are necessary.

Data prefetching is an effective technique to reduce the average memory access latency of a program. Prefetching schemes aiming for low prefetch hit times use small prefetch buffers close to the L1 cache. Since the number of slots in the prefetch buffer is limited, prefetch algorithms opt for high accuracy, where only highly predictable streams are prefetched. However, this usually translates to low prefetch coverage that results in fewer of the original program misses being correctly prefetched. When designers choose to favor high coverage, the L2 cache is the common storage for prefetches. The larger size of the L2 can tolerate inaccurate prefetches, but it can only mask a portion of the memory latency.

Realizing the need for optimizing both L1 and L2 cache miss rates, we proposed a framework for two-level prefetching in this paper. Our techniques aim to increase cooperation among the prefetchers while attempting to avert potential bus congestion and cache contention issues. Our experimental evaluation of a stride L1 prefetcher and a hybrid nextline/markov L2 prefetcher connected in parallel mode achieved 92% speedup over stride L1 prefetching technique alone. Furthermore, for an aggressive prefetcher pairing of the stride L1 prefetcher and the scheduled region L2 prefetcher, the cooperative drafting model achieved an average speedup of 13% over L2 prefetching alone (reaching 25% in the best case). Overall, we find that training the L2 prefetcher on a filtered set of confident L1 prefetches helps improve the timeliness of the L1 prefetcher while minimizing the bus congestion created by these extra L2 prefetches.

References

[1] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[2] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[3] K. Farkas and N. Jouppi. How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors? In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 78–89, January 1995.

[4] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[5] N. 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*, May 1990.

[6] W.-F. Lin, S.K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, February 2001.

[7] 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*, October 2002.

[8] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *33rd International Symposium on Microarchitecture*, December 2000.

[9] Z. Wang, D. Burger, K.S. McKinley, S.K. Reinhardt, and C. C. Weems. Guided Region Prefetching: A Cooperative Hardware/Software Approach. In *30th Annual International Symposium on Computer Architecture*, June 2003.