Journal of Instruction-Level Parallelism 13 (2011) 1-28 Submitted 10/10; published 3/11

Performance Oriented Prefetching Enhancements Using Commit Stalls

R Manikantan RMANI@CSA.IISC.ERNET.IN
R Govindarajan GOVIND@CSA.IISC.ERNET.IN
Indian Institute of Science, Bangalore, India

Abstract

Loads that miss in L1 or L2 caches, and are waiting for their data at the head of the ROB, cause significant slow down in the form of commit stalls. We identify that most of these commit stalls are caused by a small set of loads, referred to as *LIMCOS* (Loads Incurring Majority of COmmit Stalls). We propose simple history-based classifiers that track commit stalls suffered by loads to help us identify this small set of loads. In this paper we study two prefetching enhancements enabled by classifiers.

In the first enhancement, the classifiers are used to train the prefetcher to focus on the misses suffered by LIMCOS. This, referred to as focused prefetching, results in a 9.8% gain in IPC over naive GHB based delta correlation prefetcher along with a 20.3% reduction in memory traffic for a set of 17 memory-intensive SPEC2000 benchmarks. Another important impact of focused prefetching is a 61% improvement in the accuracy of prefetches. We demonstrate that the proposed classification criterion performs better than other existing criteria like criticality and delinquent loads Also we show that the criterion of focusing on commit stalls is robust enough across cache levels and can be applied to any prefetcher without any modifications to the prefetcher. We also demonstrate the positive impact that *Focused Prefetching* has in a multi-core scenario. In the case of global history based prefetchers, we demonstrate not only the applicability of focused prefetching, but also the second enhancement based on classifiers – filtering of prefetches once they are generated.

1. Introduction

In-order commit is employed in superscalar processors to ensure that the architected state of the processor is updated by instructions in program order even though instructions may be issued and executed out-of-order. The downside of in-order commit is experienced when long latency instructions and loads that miss in the cache reach the head of the ROB and wait for their completion or arrival of data. This stalls the commit of all future instructions including those which have already completed execution. Such commit stalls have a negative impact on performance. On the other hand, it is not easy to implement out-of-order commit processors as it requires expensive checkpointing mechanisms to ensure correctness [1]. Further, expensive memory checkpointing mechanisms are required to support out-of-order commit of stores.

Figure 1 shows that in the SPEC2000 benchmark suite[1], close to 60% of commit stalls are caused by loads[2]. These load stalls are experienced in spite of having a hierarchy of caches (in this case L1 and L2). Prefetching is widely used to augment the performance of caches by bringing in data into the caches before an actual demand request will be made.

1. *fma3d* is not considered as it did not run in our framework.
2. The machine configuration and simulation parameters are summarized in Section 5, and no prefetcher was used for this study.

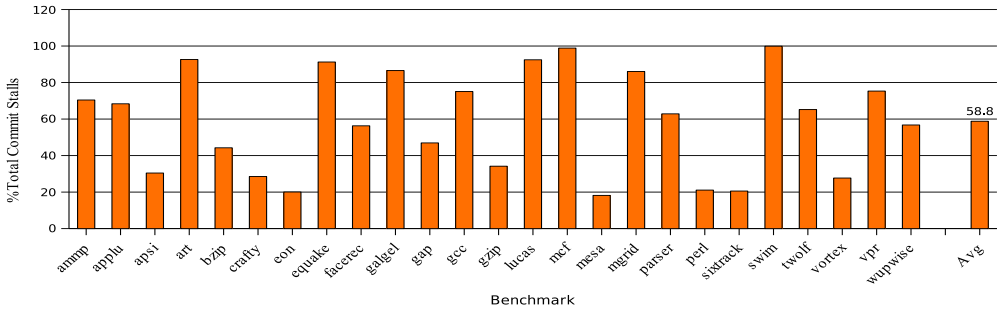


Figure 1: Fraction of commit stall cycles that can be attributed to loads.

A wide variety of prefetchers have been studied for data caches [2, 3, 4, 5, 6, 7, 8, 9, 10]. All these prefetchers are normally trained on the miss/access stream and identify useful patterns/trends among the accesses. The impact of any prefetcher on performance is based on the usefulness and the timely arrival of prefetched data. However, prefetchers can have a negative impact on performance due to increased memory traffic and pollution caused by the prefetched data in cache [11].

An analysis of the commit stalls caused by various load instructions shows that a small number of loads account for a large fraction of the commit stalls. These loads are referred to as LIMCOS (Loads Incurring Majority of COmmit Stalls). We demonstrate that simple classifiers based on history can be designed to easily identify this small set of loads. The classifiers are off the critical path and work by tracking the stalls experienced by individual loads.

In this paper, we propose *Focused Prefetching*, a mechanism that uses the classifiers to filter the training stream seen by a prefetcher, i.e., only the misses suffered by loads identified by the classifier act as the training stream for the prefetcher. The intuition behind *Focused Prefetching* is that using the limited hardware resources of prefetchers to store more information about LIMCOS can help us learn their behaviour better. By focusing on misses suffered by LIMCOS, focused prefetching eliminates misses that have a significant impact on performance. The other interesting aspect is that our method is agnostic to the prefetcher used. As our method does not change the internal working of a prefetcher, it can be applied to any of the current prefetching mechanisms.

Experimental evaluation shows that *Focused Prefetching* improves performance (IPC) by 9.8% on an average for a set of 17 memory-intensive SPEC2000 benchmarks over naive prefetching using Global History Buffer (GHB) and delta correlation prefetcher [8]. Also this gain in performance is achieved along with a 20% reduction in memory traffic and a 61% improvement in the accuracy of prefetches. In a quad-core scenario, *Focused Prefetching* improved the performance of naive prefetching by 6.0% for multi-programmed workloads. The interesting feature of our method is the applicability of *Focused Prefetching* to global history based delta correlation prefetchers [8]. These prefetchers are dependent on seeing the entire training stream to predict future accesses. Contrary to expectations, even for this class of prefetchers, filtering the training stream helps in improving the performance in terms of IPC by 4.6% on an average. Also in the context of global history based prefetchers, a more suitable alternative will be to filter the prefetches once they are generated. In this way, no useful training information is lost. Thus, for global history prefetchers, we demonstrate the application of our LIMCOS classifiers to *filter* the prefetches once they are generated. The filtering based on commit stalls improved the performance of the naive prefetcher by 4.9%. Finally, we

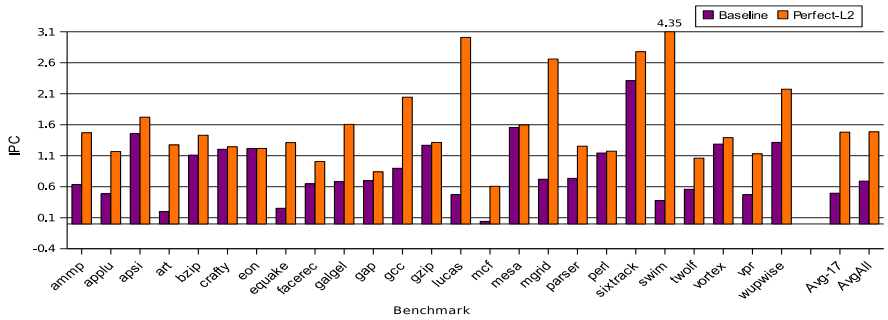


Figure 2: IPC for Baseline (No Prefetch) and PerfectL2.

demonstrate that the classification criterion of load commit stalls is better than existing criteria like either criticality [12] or load-specific criteria like delinquent loads [13]. Compared to the approaches mentioned above, our *Focused Prefetching* scheme results in an IPC improvement of 4.6% and 4.2% respectively.

In Section 2, we present the motivation behind our work. Simple classifiers to identify LIMCOS loads are discussed in Section 3. An application of the classifiers, *Focused Prefetching* is discussed in Section 4. Detailed simulation results and comparison with other schemes are presented in Section 5. A summary of related work can be found in Section 6. Concluding remarks are presented in Section 7.

2. Motivation

2.1. Memory Intensive Benchmarks

Figure 2 shows the absolute IPC for baseline and a machine with a perfect L2 cache. The machine parameters can be obtained from Section 5 and no prefetcher is used for the purpose of this study. The configuration with perfect L2 can be thought of as an 100% accurate and timely prefetch to mask all L2 misses. The potential gains in performance are an indicator of the memory intensive nature of the benchmarks. In 8 benchmarks, *apsi, crafty, eon, gzip, mesa, perl, sixtrack, vortex*, the performance improvement with a perfect L2 is very small. The remaining benchmarks show at the least 20% improvement in IPC with perfect L2. These are classified as memory-intensive benchmarks. Similar criterion has been used in earlier works [8] to identify memory-intensive benchmarks. Henceforth, we will discuss results in detail for the set of 17 memory-intensive benchmarks identified here. Fig 2 also shows the geometric mean of IPC for all benchmarks and the 17 memory-intensive benchmarks.

2.2. Commit Stalls and Individual Loads

Figure 1 shows that loads account for most of the commit stalls. We analyze it further by studying the contribution of individual loads. The fraction of commit stalls accounted by various number of static loads (from 1 to 64, in powers of two) is shown in Figure 3 for the 17 memory-intensive benchmarks.

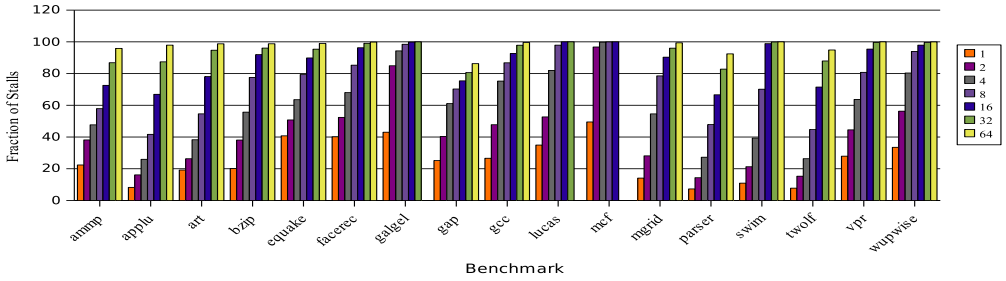


Figure 3: Individual Loads and their contribution to commit stalls: Fraction of Commit Stall accounted for by various number of static loads.

It can be observed that in all the benchmarks, only a small set of static loads account for most of the commit stalls. For instance, 10 static loads can account for anywhere between 50% (*twolf*) to 95% (*galgel,lucas,mcf,wupwise*) of the total stalls caused by loads. The only exception to this among the memory-intensive benchmarks is *applu* which requires 11 static loads to cover 50% of load commit stalls. 16 static loads as can be observed from Figure 3 account for at the least 70% of commit stalls except in *applu* and *parser*.

As the LIMCOS loads encounter commit stalls mainly due to cache misses, we carried out a limit study to identify the potential benefits that could be achieved if these loads were to hit in the L2 cache. For the purpose of the limit study, we used a profile run to identify the static loads that account for 50% of load commit stalls(LIMCOS-50). We implemented an idealized scheme, referred to as *Instant Replacement*, similar to [13], in which the static loads identified in the previous step, suffer no L2 cache misses. In other words, any data requested by the selected set of static loads is brought instantaneously into the L2 cache if it is not present in the cache. This can be thought of as an 100% accurate and timely prefetch focused on the small set of static loads.

Figure 4 shows the gain in IPC over baseline (no prefetch) for *Instant Replacement*. Applying *Instant Replacement* for the static loads in LIMCOS-50 results in a 63% gain in IPC over baseline for the memory-intensive benchmarks (the corresponding number is 44% for the entire set of 25 benchmarks studied). As *Instant Replacement* mimics 100% accurate prefetching for LIMCOS-50, the results indicate that there is a scope for significant performance gain by focusing on the LIMCOS loads.

2.3. Commit Stalls and Delinquent Loads

Previous research [13] has shown that a small set of loads account for a major fraction of the cache misses. This small set of loads is referred to as *Delinquent Loads*. A load experiences commit stalls only when it misses the L1 cache and is waiting for data to arrive from lower levels of cache or memory. Naive expectations might lead one to believe that the loads that account for most of the misses, the Delinquent loads, will account for most of these commit stalls. But this need not necessarily be the case, and a comparison between the loads accounting for commit stalls and the delinquent loads, shows only a partial overlap. This can be observed from Figure 5, which shows

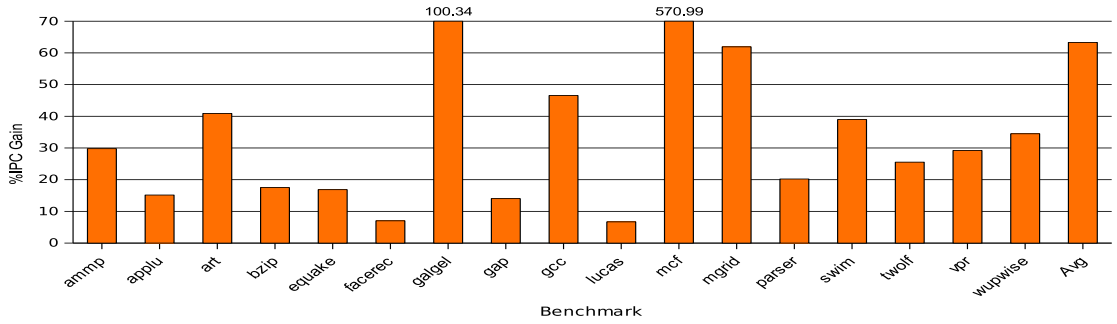


Figure 4: Gain in Performance for Instant Replacement over Baseline.

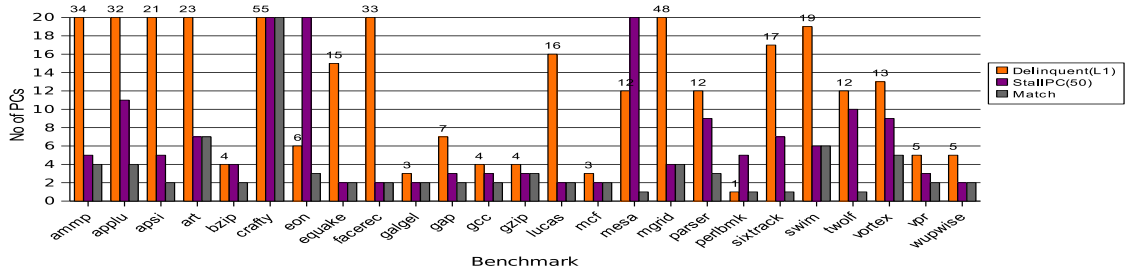
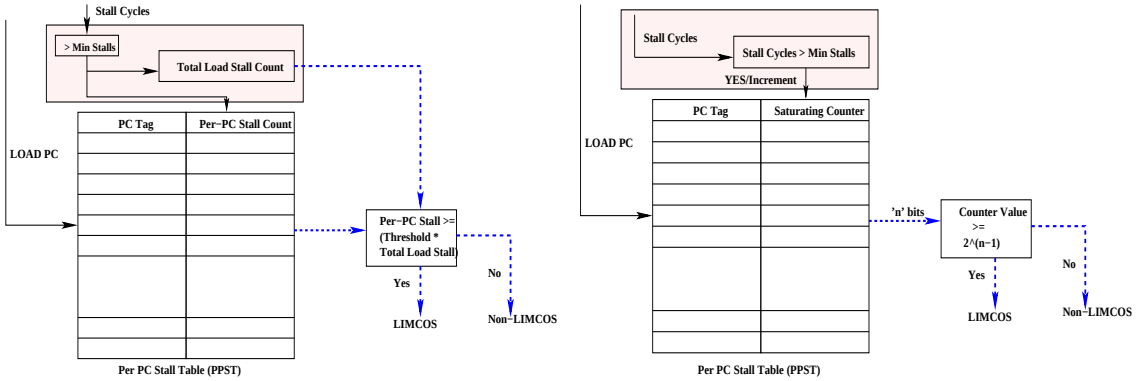


Figure 5: Relation to Delinquent Loads: Overlap between Loads accounting for 50% of stalls and 50% of misses.

the number of static loads that account for 50% of misses, the number of static loads that account for 50% of load commit stalls and the overlap between them. When more than one delinquent load miss happens in parallel, the oldest miss is held responsible for the commit stalls and the later delinquent loads might not get counted as part of LIMCOS. The lack of a perfect match between the delinquent loads and the LIMCOS loads can be attributed to the effects of Memory Level Parallelism (MLP) [14]. *Focused Prefetching* using *Classifiers*, will identify all these overlapping loads one after the other (from the oldest to the youngest) and will eliminate the stalls suffered by them. Initially *Focused Prefetching* will target the oldest among the overlapping loads and will try to remove stalls experienced by it. On addressing the stalls seen by the oldest load, the second oldest among the overlapping loads will now contribute to many commit stalls and hence will be classified as LIMCOS. Hence the claim above that overlapping loads will be addressed one after the other.

3. Classifiers

In this section, we discuss the rationale behind the design of our classifiers. Subsequently, we propose two types of classifiers and evaluate their effectiveness in identifying LIMCOS.



(a) Counting Classifier (b) Confidence Based

Figure 6: Classifier Organization.

3.1. Rationale Behind Classifiers

Figure 1 indicates that loads account for most of the commit stalls, while Figure 3 further shows that only a few static loads account for most of the commit stalls. Taking the two facts together, we conclude that a few loads should suffer commit stalls frequently. We make use of this fact to design simple history based classifiers that can identify loads causing a significant fraction of commit stalls. We propose two types of classifiers, a *Counting classifier* which uses counters to keep track of the absolute number of stalls experienced by each load and a *Confidence based Classifier*, which approximates the counts using confidence counters.

3.2. Counting Classifier

The Counting Classifier works by keeping track of the stalls experienced by the individual loads. Any load that has accounted for more than a certain fraction of total stalls seen so far is classified as one stalling frequently. Figure 6(a) illustrates the key structures and the organization of the counting classifier. There are two main operations associated with any classifier viz., *Update*, where the classifier needs to be updated when a load incurs commit stalls and *Classification*, where for a given load, the classifier needs to decide whether or not it belongs to LIMCOS.

As can be seen from Figure 6(a), the classifier is an array of counters, called Per PC Stall Table(PPST), which is tagged and indexed based on load PC. An *Update* is performed when a load that has stalled at the head of the ROB for a few cycles commits. An *Update* operation requires the knowledge of the load PC and the number of cycles of stall incurred by it. Indexing based on load PC is common to both *Update* and *Classification* operations. The operations that are specific to *Update* are shown in the shaded region of Figure 6(a). *Updates* are carried out only if the number of stall cycles encountered is greater than *Min Stalls*, a design parameter of the classifier specified in terms of number of processor cycles. This helps to reduce the number of entries required in the classifier and to avoid updates from loads that do not experience frequent stalls. In case of the stall cycles incurred by the load being above *Min Stalls*, it is added to the PPST entry of the load (identified by the PC) and is also added to the global counter which indicates the total commit stalls caused by loads. In case the load in question is not being tracked by the classifier, a new entry

is allocated in the PPST to track the stalls experienced by the load. LRU replacement is used to identify the candidate for replacement in the classifier.

The *Classification* procedure should indicate as to whether a load belongs to LIMCOS or not. The dotted lines in Figure 6(a) show the steps involved in *Classification*. The *Counting Classifier*, as mentioned above, is indexed using the PC of the load. If the load in question is not being tracked by the classifier currently, it is classified as non-LIMCOS. Otherwise, the load is classified as LIMCOS if the stall cycles in the corresponding PPST entry accounts for more than a *Threshold* fraction of the total stalls caused by the loads. Thus the counting classifier has two parameters, namely *Min Stalls* and *Threshold*.

The design with a single global counter tracking the commit stalls caused by all the loads can affect the efficiency of the classifier as new entries in the PPST will never get classified as LIMCOS due to the high value of the global counter. To overcome this, we clear the global counter and all the PPST entries periodically. This period is set as 1 million cycles for all the simulations carried out in this study. Also we wait until a reasonable amount of history is gathered before we make any attempts at classifying a load. This value is fixed as 10,000 load stall cycles. The efficiency of this design, in terms of impact on performance, is discussed in detail in Section 5.3.1..

3.3. Confidence Based Classifier

The *Confidence Based Classifier* is an approximation of the mechanism behind the *Counting Classifier*. The organization of the *Confidence based Classifier* is illustrated in Figure 6(b). The key difference is the use of saturating counters in PPST instead of counting the actual number of stalls experienced by each load. We used 5 bit saturating counters in each PPST entry. An *Update*, indicated by the shaded region of Figure 6(b), involves incrementing the confidence counter for a given load if the stall cycles caused by it is greater than *Min Stalls*. The PPST is indexed using the load PC and the replacement of existing entries, if required is carried out using LRU policy, as in the counting classifier. *Classification*, indicated by dotted lines, classifies a load as LIMCOS if the counter value is more than half of the maximum value. In the following sections, we will see that our proposed scheme, *Focused Prefetching*, attempts to eliminate commit stalls suffered by the LIMCOS loads. If the attempt is successful, LIMCOS loads will start seeing fewer and fewer commit stalls. Yet they should still be classified as LIMCOS. Hence we do not reduce the counter value if the load is classified as Non-LIMCOS. The classification mechanism based on observing the confidence value also eliminates the need for the global counter which is present in *Counting Classifier*.

While the basic principle behind the working of both the classifiers is the same, there are a few differences between the classifier designs. In the presence of focused prefetching, which is discussed in Section 4, the stalls suffered by a load identified as belonging to LIMCOS by the classifiers will be eliminated to a greater extent. Thus the dynamic instances of this static load might not incur commit stalls. Yet they still need to be identified as LIMCOS. The confidence based classifier will classify these loads as LIMCOS (as there is no decrement of confidence) and will enable focused prefetching. The counting classifier, on the other hand might not classify future instances of this load as frequently stalling as (i) the commit stalls are removed due to focused prefetching and (ii) other loads might add to the overall stalls caused by loads and the *Threshold* might not be met over a period of time. For small saturation values, the confidence based classifier

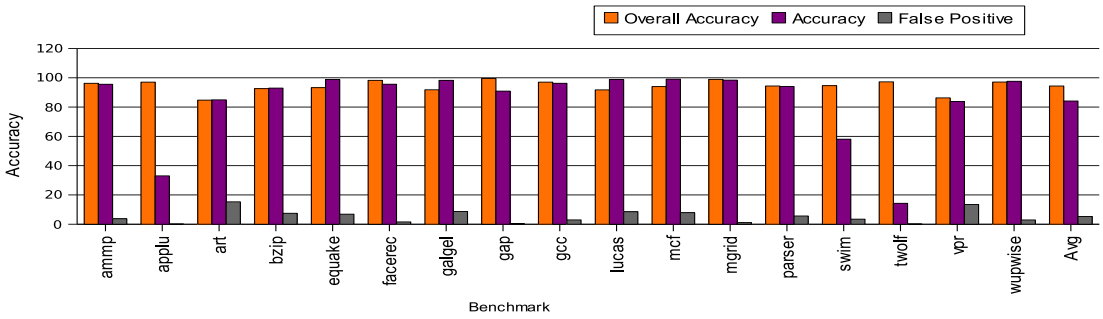


Figure 7: Accuracy of Counting Classifier: 32 Entries, Min Stalls 16 cycles, Threshold 1/32.

can learn faster and will be able to make classification decisions earlier compared to the counting classifier. The key characteristics common to the design of both the classifiers are

- The tracking of commit stalls is an operation already supported in the processors. The classifiers need to be updated only when loads suffer commit stalls.
- The update of the classifier can be kept entirely off the critical path
- As the classification decisions are based on PC of the load instruction, they can be initiated much earlier in the pipeline, once the PC is known.

3.4. Evaluating the Classifiers

In this section, the classifiers are evaluated on the basis of their ability to accurately identify LIMCOS and non-LIMCOS loads. The machine configuration used for these experiments are presented in Table 1. No prefetcher is used during these studies. The ability of the classifiers to identify correctly the loads accounting for 50% of the load commit stalls, LIMCOS-50[3] is studied. The criteria used to judge the performance of the classifiers are: (i).*Overall Accuracy*: The fraction of dynamic loads that are identified correctly as either belonging to LIMCOS-50 or not. (ii).*LIMCOS Accuracy*: The fraction of LIMCOS loads that are classified accurately. (iii).*False Positive Rate*: The fraction of non-LIMCOS loads that are wrongly identified as belonging to LIMCOS.

Figure 7 shows the *Overall Accuracy, LIMCOS accuracy* and *False Positive Rate* for the *Counting Classifier* design used in the rest of this study. The overall accuracy is 94.4% on an average for the set of 17 memory-intensive benchmarks. Further the LIMCOS accuracy is also high, (84% on an average), and the false positive rate, on an average, remains at a low 5%. In our experiments, the *Min Stalls* is kept at 16 cycles for *Counting Classifier* and 32 cycles for *Confidence Based Classifier* to enable the *Counting Classifier* to learn quickly as the counters in the PPST are cleared periodically after every million cycles.

Figure 8 shows the *Overall Accuracy, LIMCOS accuracy* and *False Positive Rate* for the *Confidence Based Classifier* design used in the rest of this study. While the Overall Accuracy (84%) is

3. The trends observed were similar for LIMCOS-80, where loads accounting for 80% of the load commit stalls are treated as part of LIMCOS.

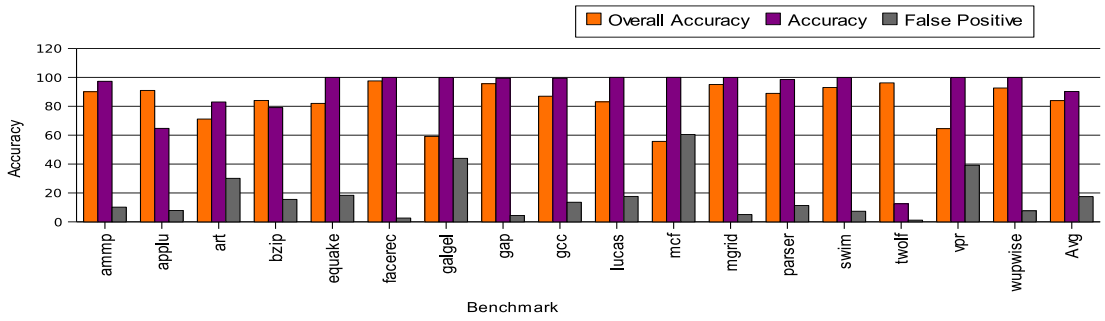
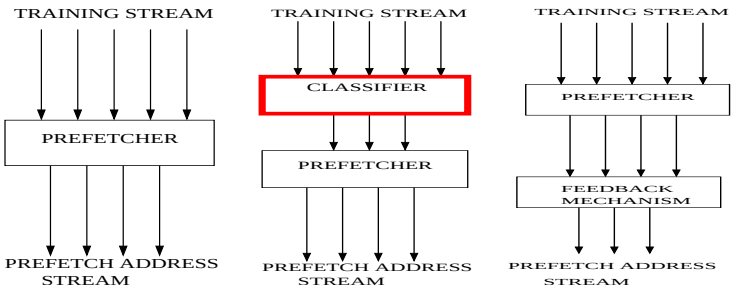


Figure 8: Accuracy of Confidence Classifier: 32 Entries, 8 Way Associative, Min Stall 32 cycles.



(a) Current (b) Focused Prefetching (c) Feedback Directed

Figure 9: Focused Prefetching and other prefetch mechanisms.

slightly lower compared to the *Counting Classifier*, the quick learning allows the confidence based classifier to achieve high LIMCOS Accuracy of 90%. The flip side of the quick learning and the not reducing the confidence values can be seen by the relatively higher false positive rate of 17%. The interesting aspect to note is the reasonably high overall accuracy achieved by the classifiers in-spite of using only 32 entries.

4. Focused Prefetching

Focused prefetching is an application of the classifiers mentioned in Section 3.1.. Focused prefetching is a filtering mechanism that helps any prefetcher to focus more on misses that are likely to have a significant impact on performance.

Any existing prefetcher, as shown in Figure 9(a), is made up of three components – the main prefetching algorithm, the input to it which is normally a stream of misses and the output which is a stream of prefetch addresses. The prefetching algorithm identifies trends in the input stream and generates the prefetch stream which contains the addresses that are likely to be accessed in the future.

In *Focused Prefetching*, the thrust is in filtering the training stream seen by the prefetcher. As shown in Figure 9(b), given any prefetch algorithm, we use a classifier to filter the training stream so

that the core of the prefetcher sees only the misses caused by LIMCOS loads. One of the two classifiers proposed in Section 3 could be used to implement *Focused Prefetching*. The rationale behind this filtering is that, by definition of LIMCOS loads, eliminating the misses suffered by the loads identified by the classifier will lead to lesser commit stalls and improved performance. Also seeing only a part of the training stream, will allow the prefetcher to use its hardware resources efficiently and improve the accuracy of the prefetches. The improved accuracy and generating prefetches in response only to a subset of the misses translates into lesser number of wasted prefetches being generated. This also alleviates the pressure on the memory and reduces the memory traffic caused by naive prefetching. *Focused Prefetching* is oblivious to the underlying prefetch mechanism and hence has a wide applicability.

An important aspect to consider in *Focused Prefetching* is the *timing* of the *Classification* requests to the classifier. The outcome of the *Classification* step decides whether or not the miss will form a part of the input stream to the prefetcher. If the prefetcher is associated with a cache level where the load PC information is available, the *Classification* request could be made once a miss is suffered. At caches closer to memory, where load PC information is generally not available [8], the *Classification* request has to be made earlier in the pipeline and the result has to be propagated along with the load request. As the classifiers are indexed based on load PC, the classification request could be made once the PC is known. In our simulations, the classification request is made earlier in the pipeline, once the instruction is identified as a load.

A recent research in eliminating the harmful effects of prefetching and deriving the maximum benefit out of it is *Feedback Directed Prefetching (FDP)* [11]. FDP, as shown in Figure 9(c), filters the prefetches once they are generated based on the prefetch accuracy, timeliness and pollution caused by the prefetcher. FDP achieves this by controlling the prefetch degree. Further, FDP is a reactive mechanism and is oblivious to the importance of the misses eliminated by the prefetcher. FDP is orthogonal to Focused Prefetching and can complement our scheme to improve its performance.

5. Results

5.1. Simulation Details

The simulation framework used in this study is built on top of the *sim-alpha* simulator [15]. The machine model and other relevant parameters are presented in Table 1. Each level of cache has 32 MSHRS [16] out of which 16 are reserved for prefetches. Regular accesses are given priority over prefetches. We used the early single simulation point [17] for all our simulations. The interval size considered is 100 million instructions.

Most of the detailed evaluation is carried out for prefetching at the L2 cache. For this purpose, we consider a per-PC Delta correlation prefetcher built on top of Global History Buffer (GHB) [8]. The prefetcher is made up of two structures, a *Global History Buffer*, which holds the most recent misses in FIFO order and an *Index Table* which chains the misses that share the same characteristics together. In this study, we use the *Index Table* to chain together misses that were caused by the same load instruction. The per-PC delta correlation prefetching mechanism uses delta pairs to decide the prefetch addresses. When a miss occurs, the two most recent deltas (differences between the 3 most recent misses) are computed. The miss history is searched backwards for a match with the delta pair computed above. Once a match is found, for a prefetch degree of 8 assumed in this study, the next 8 deltas in the per-PC miss stream following the delta pair are used to generate the prefetch addresses. The prefetching mechanism of Delta correlation is used as it is shown to be one of

Fetch/Issue/Commit Width	8
ROB/LQ/SQ	128/32/32 Entries
Int ALU/Mult	6/2
FP ALU/Mult	6/2
Branch Predictor	21264's Predictor, 32 Entry RAS
Memory Hierarchy Memory Latency	L1 DCache - 32KB, 4 Way , 32 Byte linesize, 1 cycle Unified L2 - 1MB, 8 Way, 64 Byte linesize, 12 cycles All the caches have 32 MSHRs Minimum 225 cycles
Prefetcher Prefetch Degree	At L2 - 512 Entry 16 Index GHB Per PC Delta Correlation 512 Entry 256 Index was also evaluated for Baseline 8
Counting Classifier Confidence Classifier	32 Entries, Lower Limit 16 and Threshold 1/32 32 Entries, 8 Way Associative, Lower Limit 32

Table 1: Machine Configuration.

the best performing prefetch algorithm[8]. Though we evaluated focused prefetching with a GHB containing 16 Index table entries (capability to chain together miss stream of 16 loads), as we focus only on a small set of PCs, for fairness, we compared it with a naive prefetcher that uses a GHB with 256 index table entries. The classifier configurations opted for here are the ones that are evaluated in Section 3.

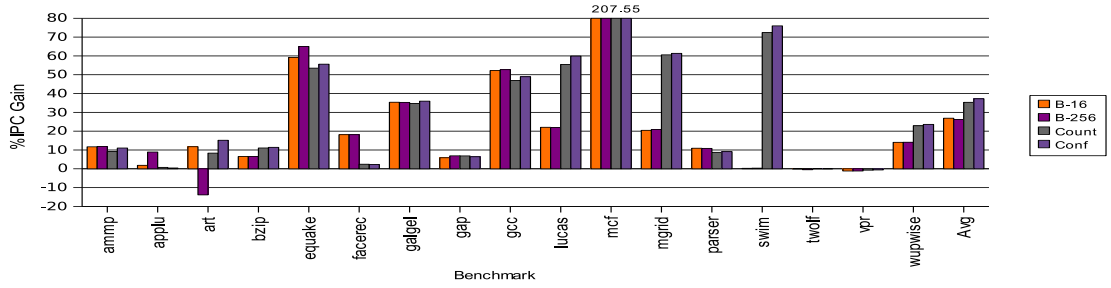


Figure 10: Performance gains of GHB with PC-Delta Correlation and Focused Prefetching over a Baseline with no prefetcher.

5.2. Performance and Traffic Gains with Focused Prefetching

In this section, we study the performance of *Focused Prefetching* when applied to a GHB based per-PC delta correlation prefetcher which tracks L2 misses and brings the prefetched data into L2.

Figure 10 shows the improvement in IPC obtained by focused prefetching and naive prefetching over no prefetching. In this figure, B-16 and B-256 stand for baseline prefetching with 16 and 256 *Index Table* entries in the GHB respectively. Results are shown for *Focused Prefetching* using both the classifiers discussed in Section 3. *Focused Prefetching* uses an *Index Table* with 16 entries. It can be observed that in most of the benchmarks, *Focused Prefetching* results in performance improvement over no prefetching and naive prefetching (baseline prefetching). On an average[4], *Focused Prefetching* with *Confidence Based Classifier* results in an IPC gain of 37.2% over no prefetching and 9.8% over naive prefetching (B-256). Between the confidence based and counter based classifiers, the confidence based classifier results in a higher IPC gain. This can be attributed to the relatively higher *LIMCOS Accuracy* of the confidence based classifier as shown in Section 3. Also the gain in IPC over B-16, 8.6% using *Confidence Based Classifier* and 7.3% using *Counting Classifier* indicates that intelligent filtering carried out by *Focused Prefetching* is better than any naive filtering achieved by having lesser number of *Index Table* entries.

Benchmarks *twolf* and *vpr* gain very little improvement in performance with any prefetching. In *lucas, mcf, mgrid, swim* and *wupwise* the effect of *Focused Prefetching* over naive prefetching is significant. On the other hand, in benchmarks like *applu, equake* and *facerec, Focused Prefetching* suffers minor performance degradation compared to naive prefetching. Especially in *facerec*, where focused prefetching is relatively unhelpful, focusing on the PCs identified by the classifiers results in a decrease in the number of prefetches by 83% and the number of useful prefetches[5] is brought down by 75%. This results in a drop in the performance compared to naive prefetching.

Figure 11 shows the reduction in the number of prefetches generated by *Focused Prefetching* compared to B-256. On an average, for the confidence based classifier, the number of prefetches generated goes down by 50% while the number of useful prefetches goes down by 26.3% (not shown in figure). In spite of this reduction in the number of prefetches, focusing on the loads in LIMCOS leads to a 9.8% gain in performance over naive prefetching. This confirms the benefits of focusing on the LIMCOS loads. Using the counting classifier results in a performance gain of 8.3% over naive prefetching despite the fact that the number of prefetches generated went down by 52.4%.

The other intended benefit of focused prefetching is the improved ability to learn trends in the filtered miss stream and generate more useful prefetches. We use the metric *Prefetch Accuracy*, which is defined as the fraction of useful prefetches among the total prefetches generated [11]. The gains in accuracy over B-16 and B-256 for *Counting Classifier* and*Confidence Based Classifier* are shown in Figure 12. *Focused Prefetching* leads to a 61% improvement in the prefetch accuracy compared to naive prefetching. All the benchmarks, even those where *Focused Prefetching* did not result in a major gain in performance, showed a gain in accuracy as a result of employing *Focused Prefetching*.

Accuracy in prefetching and focusing on a subset of misses leads to a reduction in the number of wasted prefetches, thereby saving valuable memory bandwidth. Figure 13 shows the reduction in the memory traffic measured in terms of the number of bytes transferred. It is important to consider the entire traffic rather than just the prefetch traffic as the pollution effects of prefetching can increase the miss traffic. The average reduction in memory traffic experienced is 20.3% using the confidence based classifier and 20.2% using the counting classifier. All the benchmarks showed a reduction in the memory traffic on employing *Focused Prefetching*. All the results together indicate that focusing on the small set of LIMCOS loads is beneficial to performance. In short, *Focused Prefetching*

4. We use arithmetic mean in this paper, unless specified otherwise.
5. prefetches servicing a demand access.

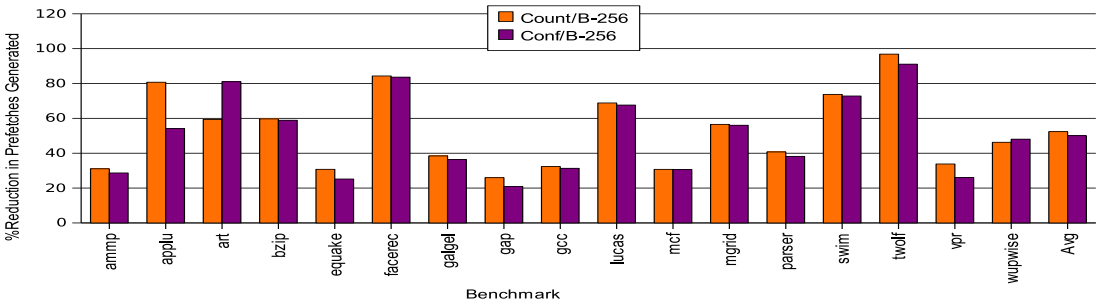


Figure 11: Reduction in the Number of Prefetches Generated by Focused Prefetching.

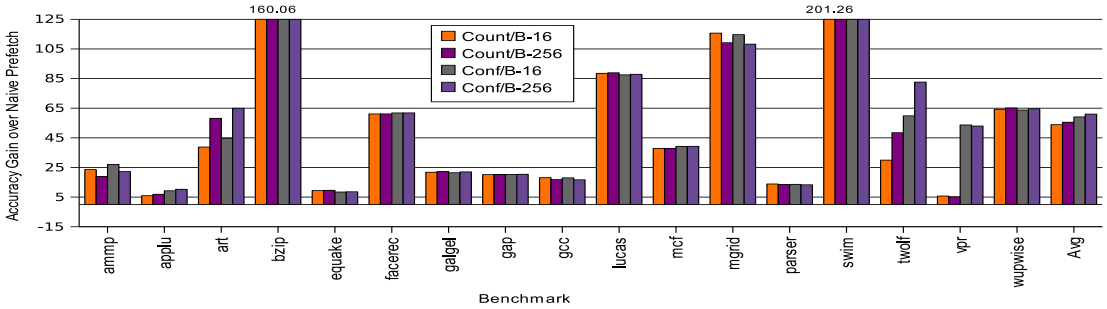


Figure 12: Improvement in prefetch accuracy due to Focused Prefetching.

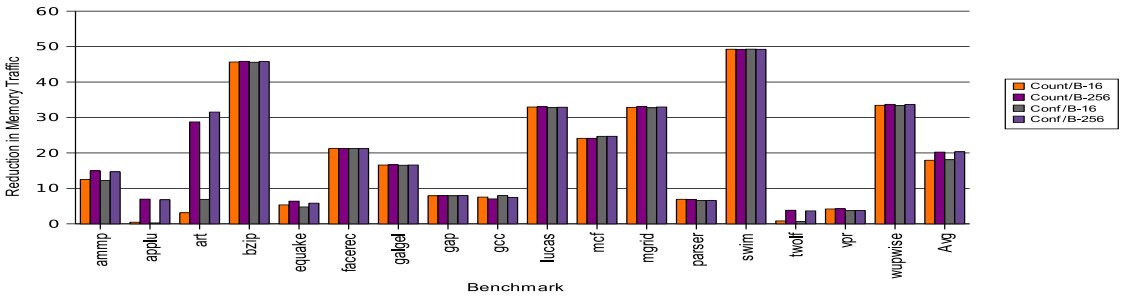


Figure 13: Reduction in Memory Traffic by employing Focused Prefetching.

enables one to eliminate the misses that matter and achieves more performance by virtue of more relevant prefetches.

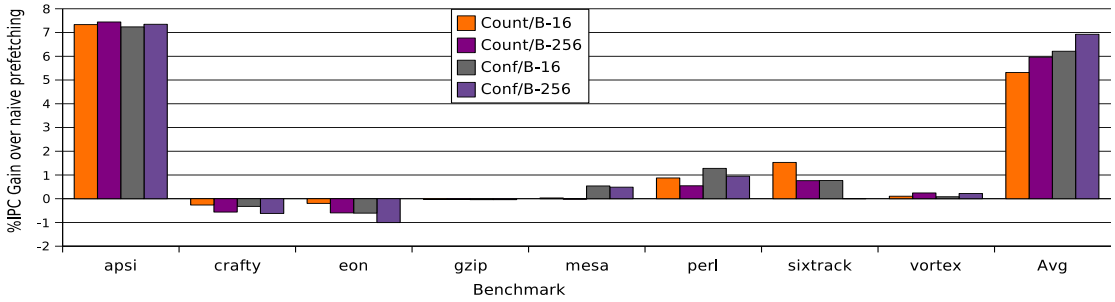


Figure 14: Gain in performance with Focused Prefetching for benchmarks not discussed in detail.

For completeness, we also show the performance gains experienced for the remaining benchmarks, except *fma3d* which did not run in our framework. The gains in performance over B-16 and B-256 by employing focused prefetching are shown in Figure 14. In *apsi*, *Focused Prefetching* results in an IPC improvement of nearly 7%. Only for *crafty* and *eon*, there is a marginal performance degradation (less than 1%) compared to naive prefetching. On an average, for all the 25 benchmarks, the gain in performance over naive prefetching is 7% for confidence based classifier and 6% using counting classifier as shown by the last set of bars in Figure 14. For the entire set of 25 benchmarks, the memory traffic reduces by 22.8% for confidence based classifier and by 23.3% for counting classifier. The interesting thing to note is that even in benchmarks that are not sensitive to memory performance, there is a substantial reduction in memory traffic by employing *Focused Prefetching*.

5.3. Classifier Design

The criterion used by counting classifier to classify a load is same as the one used to define LIMCOS, while confidence based classifier provides an approximation of the same. But the confidence based classifier is more suited to a hardware implementation. While the confidence based classifier design performed as well as the counting classifier, the former can be thought of as the ideal design choice for hardware implementation only if it manages to perform as well as the best performing counting classifier. In order to establish this, we study the design space of counting classifiers.

5.3.1. Classifier Parameters

Performance of the counting classifier is dictated not just by its *size*, but also by the other two design parameters, *limit* and *threshold*, described in Section 3. We give a brief description of the three important parameters: *size* determines the number of entries in the classifier, *limit* sets the lower limit for a commit stall to cause an update in the classifier and *threshold* is used in the classification step to specify the fraction of total stalls a load should have accounted for to be classified as LIMCOS. Though the parameters are used for different purposes in a classifier, it can be seen that they are not totally independent. For instance, with a higher value for *size*, *limit* can have a lower value without affecting the ability of the classifier adversely. Similarly, a higher value for *limit*, for instance, will mean that the classifier sees a lot less stalls during the *Update* step. This in turn might allow even

Parameter	Range of Values
Size	16, 32, 64 Entries
Limit	8, 16, 32 Cycles
Threshold	1/8, 1/16, 1/32 of Total Stalls

Table 2: Counting classifier design space.

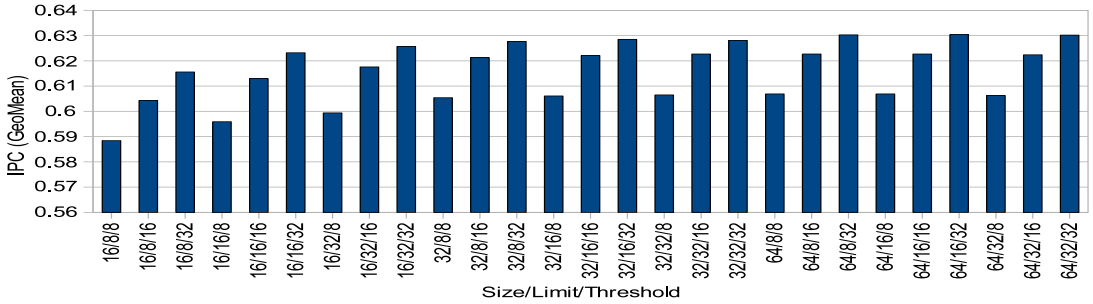


Figure 15: IPC Gain for various classifier designs.

higher *threshold* values to be satisfied during *Classification*. We varied the three design parameters associated with the counting classifier and studied their effect on the program performance. Three possible values were considered for each of the parameter resulting in 27 configurations. The values considered for the study are shown in Table 2.

Figure 15 shows[6] the geometric mean of IPC for the memory-intensive benchmarks for the various configurations of the counting classifier. Increasing the size of the classifier helps in improving performance. In comparison, the parameter *limit* has less of an impact. The impact of *limit* on performance is higher at smaller classifier sizes like 16 entries than at larger sizes like 64 entries. This is along expected lines as in a majority of the benchmarks, a few loads account for most of the commit stalls and the classifier uses a LRU replacement policy in PPST. The impact of threshold remains significant across various classifier sizes and for different values of *limit*. In general, for a given classifier, the best performance is obtained by a higher value for *limit* during the *update* operation and by relaxing the *threshold* value during the *classification* step. The other interesting thing to note is that by appropriately adjusting the *limit* and *threshold*, as mentioned above, the impact of *size* on performance can be made minimal. This can be observed from the fact that with a *limit* of 32 and *threshold* of 32, there is only minimal difference between the performance of classifiers containing various number of entries.

5.3.2. Data Retention in Classifier

The other design decision taken with respect to the counting classifiers is that of clearing the classifiers every million cycles to avoid retention of stale data. In the absence of such a clearing, with

6. The a/b/c labels shown along the X-axis in Figure 15 give the values for size, limit and (1/Threshold) respectively.

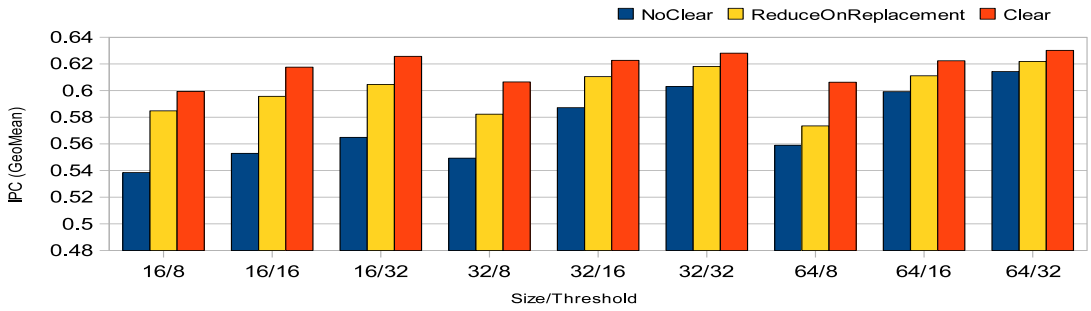


Figure 16: Stale Data in Classifier: Comparing the various alternatives at different design points. The parameter *limit* is fixed at 32 cycles.

a fixed value for threshold, the absolute number of commit stalls a load should have suffered to be classified as LIMCOS keeps increasing steadily with each *Update* operation. This will result in prefetch opportunities being ignored. This section shows that clearing the classifier entries periodically gives the best possible performance.

We considered three possible design choices for the classifier:

- *Clear*: This is the scheme used so far, in which the PPST is cleared after a fixed interval (1 million cycles in the study). The interval size didn't make much of a difference to the performance unless it was too low or too high.
- *NoClear*: In this design, the contents of classifier are modified only through the *Update* operation of the classifier.
- *ReduceOnReplacement:* As the total stall cycles suffered by each load tracked by the classifier is associated with the PPST entry allocated for it, the global stall count could be decremented on each PPST replacement by bringing the counter value down by the stall cycles stored in the replaced PPST entry.

The difference in performance gains between *Clear* and *NoClear* can be used to observe the effect of stale history in the classifiers. *ReduceOnReplacement* on the other hand lets the classifier to make the *Classification* decision only based on the current contents of the classifier. Once an entry in the PPST of the classifier gets replaced, it stops influencing the future *Classification* decisions in *ReduceOnReplacement*.

Figure 16 shows the performance of the various retention schemes at various points in the classifier design space determined by the three parameters viz., *Size*, *Limit* and *Threshold*. As *limit* had the least impact among the three parameters discussed in the previous section, to improve readability, we fix *limit* at 32 cycles while varying the other two parameters through the set of possible values listed in Table 2. The graph in Figure 16 shows the geometric mean of IPC for the set of 17 memory-intensive benchmarks. The key observations that can be made are: (i) Not clearing the stale entries in classifier, as is done in *NoClear* affects the potential performance gains, as can be

seen from the fact that it is the worst performing configuration of all.(ii) With a larger classifier, the impact of retaining all the history is mitigated to an extent.(iii) The configuration used so far in the paper, *Clear* is the best performing configuration of all.

The gains of *Clear* over *ReduceOnReplacement* could be attributed to the following reasons: (i) some of the stale entries which have seen a lot of stalls might take longer to get replaced from the classifier (ii) during some relatively stable sections of execution, the total stalls seen so far might reach high enough values that it might be harder to meet the criteria to be classified as LIMCOS. This is further substantiated by the fact that relaxing the threshold value improves the performance of *ReduceOnReplacement*.

The impact of *size* and *threshold* remain similar to what was observed for the *Clear* configuration in the previous section. Though not shown here, the impact of *limit* was higher for the *NoClear* configuration especially when the classifier had lesser number of entries. Also it is be noted that, the configuration used so far in the paper is relatively more oblivious to the classifier design parameters compared to the other schemes. Also periodic clear enables even smaller classifiers to perform as well as larger classifiers using *NoClear* or *ReduceOnReplacement* schemes. To summarize, the best possible counting classifier design was adapted in this study and the ability of the confidence based classifier to match its performance indicates that using a simpler design (confidence based classifier) or a smaller sized classifier (counting classifier with just 16 entries) can result in significant performance gains in performance and reduction in memory traffic.

5.4. Relation to Other Criteria

In this section, we present quantitative comparisons with two of the most closely related criteria for *Focused Prefetching* viz., *criticality* [12] and *delinquent loads* [13].

5.4.1. Criticality

Critical loads are defined as the loads that together with other critical instructions decide the overall execution time of the program. Earlier work has attempted to tailor prefetching schemes targeting critical loads [18]. The implementation was dependent on a set of heuristics like load leading to a load miss or branch misprediction and measuring the number of instructions issued after the load to identify the critical loads. However, such works report a significant loss in performance compared to naive prefetching for the L2 cache. For the purpose of this study, we identify critical loads using the much rigorous criteria of criticality suggested by Fields [12]. The methodology proposed in [12] works by constructing a graph where the edge weights are the delay incurred by an instruction at various stages in the pipeline waiting for true dependencies and resource constraints to be resolved. The longest path in this graph, known as the critical path, accounts for the entire execution time. Any delay to instructions in the critical path, the critical instructions, will add to the execution time of the program.

During simulations, we observed that instead of focusing on critical loads (including both hits and misses), it is better from a performance point of view to focus on the static loads that account for a large fraction of the critical misses. This is a subtle but significant difference compared to the earlier work. Thus, to implement *Focused Prefetching* with criticality as the criteria, we use the definition of Fields [12] to identify a set of static loads that account for most of the critical misses suffered at L2 cache.

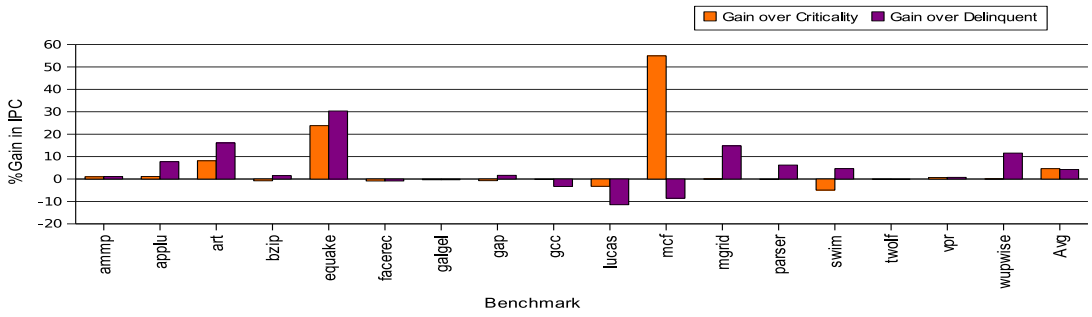


Figure 17: Performance Gains of Focused Prefetching over Criticality and Delinquent Loads.

5.4.2. Delinquent Loads

Section 2 showed that there is a partial overlap between delinquent loads [13] and LIMCOS.

5.4.3. Performance Comparison

As critical loads are identified accurately using an off-line analysis, for fairness and accuracy purposes we do not use a dynamic classifier and use profile runs to identify the loads matching the various criteria. The profile and actual runs use the same input data and are run for 100 million instructions at the simulation point [17]. The machine configuration used in the profile runs is same as the one shown in Table 1. However, no prefetcher is used in the profile runs. For each benchmark, we identify the set of static loads that account for 50% of commit stalls(LIMCOS-50). An equal number of static loads that account for most of the critical L2 misses are also identified. Similarly, one more profile run is used to identify an equal number of delinquent loads. As *Focused Prefetching* in this case eliminates the misses suffered by the static loads identified above, for fairness, it is imperative to consider same number of loads for different criteria. LIMCOS-50 is used to determine the number of loads as the criterion of commit stalls required the least number of loads to achieve 50% coverage. We implemented *Focused Prefetching* at L2 cache to focus and eliminate the misses suffered by these set of static loads identified using the three different criteria. The prefetcher used is the same prefetcher considered so far in the study

Figure 17 gives the performance improvement achieved by commit stall criterion over the other two. On an average, the gain in performance for commit stall based focused prefetching over criticality based focused prefetching is 4.6% while the gain over delinquent load based focused prefetching is 4.2%. Though there are a few benchmarks, where either criticality or delinquent loads seems to be the better criteria, in a majority of the benchmarks, focusing on commit stalls gives the maximum benefit. The only exception seems to be *lucas* where the profile based identification of commit stalls is not as efficient as other criteria. But in *lucas*, the classifiers perform well at run time as less than 10 loads account for 95% of the commit stalls, resulting in an IPC gain of 25.6% over B-256.

5.5. Different Prefetchers and Cache Levels

We apply *Focused Prefetching* to L1 Data Cache by filtering the training stream seen by a stride prefetcher. This also allows us to study the effectiveness of *Focused Prefetching* with a different

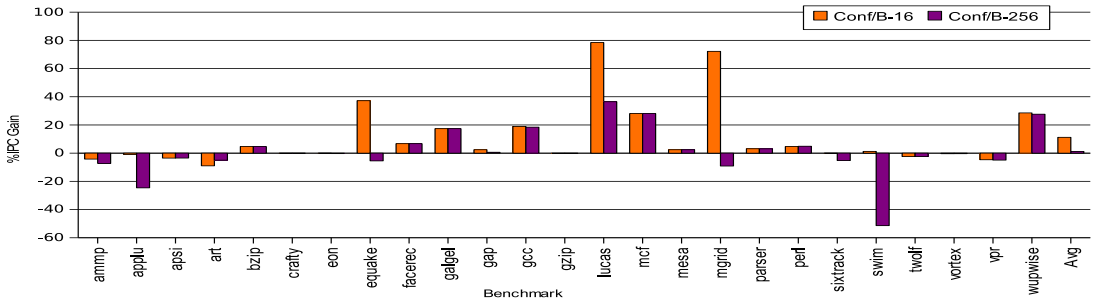


Figure 18: Performance Gains provided by Focused Prefetching at L1.

prefetch algorithm. The stride prefetcher has a per-PC stride detection mechanism and a confidence mechanism of waiting for the same stride to appear more than once in succession before issuing prefetches. The prefetch degree is set at 8. Some of the recent works on prefetching have used the stream buffer [6] as one of the prefetch mechanisms. We opted for the stride prefetcher instead of stream buffer as research shows that the performance of stream buffer improves by using a per-PC stride [3] or Markov prefetcher [19] along with it. The delta correlation predictor used earlier can be thought of as an approximation of the Markov predictor [5].

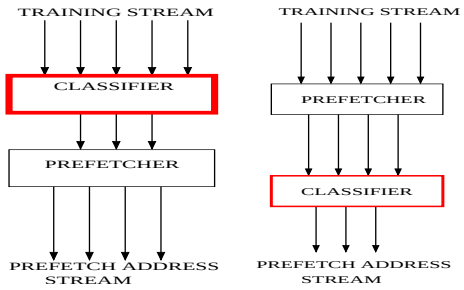
We studied *Focused Prefetching* with a confidence based classifier for the L1 cache. The performance gains over naive prefetching with an ability to track 16 per-PC Strides and 256 per-PC strides are shown in Figure 18. For the set of 25 SPEC benchmarks[7], there is a 11% gain in performance over B-16 with a 2.3% reduction in memory traffic. In a significant number of benchmarks (10 out of 25), there is at the least 5% improvement in IPC over naive prefetching. In *lucas* and *mgrid*, the gain in IPC is more than 70%. Compared to the more aggressive B-256, the IPC gain is reduced and is only 1.2%. Nonetheless, there is a gain over the naive prefetcher and the memory traffic reduces by 8.2%.

5.5.1. Global History Based Prefetching

An important class of prefetchers for which the applicability of *focused prefetching* needs to be studied is the global history based prefetchers [8]. *Focused Prefetching* works by filtering the history seen by the prefetch algorithm. While filtering the training data on the basis of PC is suited to Per-PC history based prefetchers that have been studied so far in the paper, it can potentially have negative impact on the performance of global history based prefetch mechanisms. Global history based prefetchers rely on the regularity that is present in the unfiltered miss stream seen at a cache level. Filtering out a part of history might result in a loss of useful correlation information in a global history based prefetcher. To study the suitability of focused prefetching to such global history based mechanisms, we apply it to the global history based variant of the delta correlation prefetching [8].

An application of classifiers that might be more suited in this case is filtering of prefetches once they are generated by a global history prefetcher. Figure 19(b) illustrates conceptually the organization of prefetcher and the classifier to achieve filtering of prefetches. As can be seen from

7. The full set of 25 benchmarks is used as all of them are sensitive to L1 cache misses.



(a) Focused Prefetching (b) Filtering Prefetches

Figure 19: Global History Prefetchers: Focused Prefetching and using classifiers to filter the prefetches.

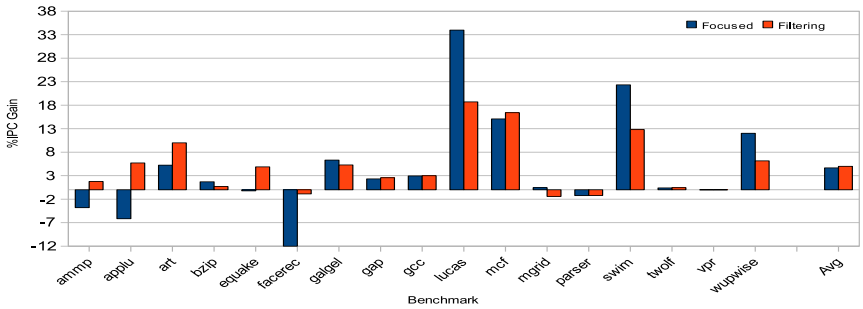


Figure 20: Global History Based Prefetching: Performance gains obtained by focused prefetching and filtering, applied to global history prefetchers, over unmodified baseline prefetcher.

Figure 19, the classifier in the case of filtering is positioned after the prefetch algorithm, at its output, unlike *Focused Prefetching*, where the classifier is positioned before the prefetcher. Hence, the prefetcher, in the case of filtering, sees all the training data and does not lose out on useful correlation information. The classifier is used to decide when to trigger prefetches. In this case, only misses suffered by loads in LIMCOS trigger prefetches.

We used a confidence based classifier for the purpose of this study. The configuration of the classifier remains the same as in the previous studies. The prefetch mechanism of global history based delta correlation was implemented on top of a GHB with 512 entries and 256 index table entries. A prefetch degree of 8 was used. Figure 20 shows the gain in IPC over baseline unmodified prefetching for focused prefetching and filtering, both the schemes implemented on top of the baseline global history based delta correlation prefetcher.

On an average the gains obtained by both the schemes are similar, 4.66% for focused prefetching and 4.99% for filtering over naive prefetching. While naive prefetching resulted in IPC gains of 20.12% over no prefetching, focused prefetching and filtering achieved 26.12% and 26.20% gains in IPC respectively. The negative effects of filtering the history, as is done in focused prefetching, is

seen primarily in *ammp*, *applu* and *facerec* which lose performance compared to naive prefetching. However, there is still a gain in performance over no prefetching even for these three benchmarks. The loss in performance for *facerec* is on the higher side, where a reduction in the prefetch opportunities hurts the performance, similar to the per-PC history case discussed earlier. It is also interesting to note that even filtering of generated prefetches affects performance negatively in the case of *facerec*. Other benchmarks where filtering affects performance negatively are *mgrid*, *parser* and *vpr*. While *ammp*, *applu* and *equake* favour filtering over focused prefetching, the situation is reversed in the case of *lucas*, *swim* and *wupwise* where significant gains are to be had by employing focused prefetching even along with a global history prefetcher. On a whole, focused prefetching retains the advantages provided by it even in the case of global history prefetchers though the impact is reduced in this case. Filtering generated prefetches using classifiers demonstrates yet another use of classifiers to enable performance aware prefetching.

5.6. Focused Prefetching in Multi-Cores

The evaluation carried out so far focused on the performance of individual applications run in a stand-alone fashion. But modern processors are multicore machines with typical usage scenario consisting of multiple programs being run in parallel. In such a scenario, the performance of shared resources in the memory subsystem – caches, memory bandwidth – become crucial to the performance. *Focused Prefetching* can help in relieving some of the pressure on the memory subsystem by improving the prefetch efficiency. Hence in this section, we evaluate the performance of *Focused Prefetching* in the context of multicores running multi-programmed workloads.

We use M5 [20] simulator to study the performance impact of *Focused Prefetching* in multicores[8]. We simulate quad-core machines running 4 benchmarks in parallel. The benchmarks are selected from SPEC2000 and SPEC2006 benchmark suites. We are not studying schemes were other cores are used to prefetch data for the main thread (Helper threads and the like) [21]. The CPU and L1 cache parameters are same as that used in the rest of the paper. The L2 cache in this case is shared among the 4 cores. We simulate a 8MB 32 way associative L2 cache for the purpose of this study [9]. The machine uses *Alpha* ISA. Each core uses the same confidence based classifier as the rest of the study. The Quad-core workloads used in this study are listed in Table 3. We use harmonic mean speedup(SMT speedup) [22] to summarize the performance of the multi-programmed workloads. Harmonic mean speedup of N applications being run in parallel is defined as N/$\sum(StandAloneIPC_i/IPC_i)$. Here IPC_i is the performance of the i^{th} benchmark in the workload in terms of its IPC. StandAloneIPC is the IPC observed when the benchmark is run alone.

The prefetcher in this study is situated at the shared L2 and we use a Per-PC delta correlation prefetcher. All the programs were fast forwarded for 10 billion instructions, with the last 1 billion instructions used to warmup the processor structures. We simulated each workload in detail for 2 billion cycles and report the observed performance. Figure 21 shows the harmonic mean speedup of naive prefetching and *Focused Prefetching* normalized to that of a machine with no prefetching. The naive baseline prefetching provides a speedup of 22% over no prefetching (Geometric mean) while *Focused Prefetching* improves the performance by 29% compared to no prefetching. In all cases, except workload Q6, it can be seen that prefetching is beneficial to performance. In the case

8. The sim-alpha [15] simulator used in the rest of the study did not have the support to evaluate multicores.
9. We observed similar performance trends with 2MB and 4MB caches.

Q1	(300.twolf, 256.bzip2, 183.equake, 482.sphinx3)
Q2	(178.galgel, 179.art, 471.omnetpp, 410.bwaves)
Q3	(434.zeusmp, 183.equake, 459.GemsFDTD, 470.lbm)
Q4	(462.libquantum, 187.facerec, 183.equake, 171.swim)
Q5	(183.equake, 482.sphinx3, 189.lucas, 470.lbm)
Q6	(470.lbm, 168.wupwise, 171.swim, 300.twolf)
Q7	(179.art, 470.lbm, 183.equake, 171.swim)
Q8	(168.wupwise, 482.sphinx3, 171.swim, 470.lbm)
Q9	(178.galgel, 459.GemsFDTD, 471.omnetpp, 179.art)
Q10	(187.facerec, 179.art, 171.swim, 410.bwaves)
Q11	(471.omnetpp, 459.GemsFDTD, 301.apsi, 300.twolf)
Q12	(183.equake, 187.facerec, 171.swim, 470.lbm)
Q13	(300.twolf, 171.swim, 482.sphinx3, 459.GemsFDTD)
Q14	(459.GemsFDTD, 179.art, 183.equake, 470.lbm)

Table 3: Quad-Core Workloads.

of Q6, while benchmarks 470.lbm and 171.swim gain in performance, performance degradation observed in the case of 168.wupwise and 300.twolf affects the harmonic speedup observed with prefetching. This primarily happens as harmonic speedup also takes into account fairness along with performance gain [22]. In fact, with naive prefetching Q6 shows a gain of 24.6% and 13.3% over no prefetching when the performance is summarized using other metrics like throughput [22] and weighted speedup [22] respectively. With *Focused Prefetching*, Q6 shows 26.7% and 14.3% improvement over no prefetching using throughput and weighted speedup metrics respectively. For the entire set of workloads, naive prefetching improves performance over no prefetching by 24.4%(Throughput) and 24.2%(Weighted-speedup). Using *Focused Prefetching*, the performance improvement over no prefetching are 31.9%(Throughput) and 31.6%(Weighted-speedup) [10].

Similar to the case of single cores, *Focused Prefetching* can benefit multicores too and provides an average (Geometric Mean) speedup of 5.7% over naive baseline prefetching in terms of harmonic/fair speedup metric. In terms of other metrics like throughput and weighted speedup, the gains provided by focused prefetching over naive baseline prefetching are 6.0% and 5.9% respecitvely (Geometric Mean). Significant improvement in performance is seen in workloads Q7 and Q12, where *Focused Prefetching* improved the performance of all the individual benchmarks that constitute the workload. Noticeable gap in performance between naive and *Focused Prefetching* is seen only in the case of workload Q10 where *Focused Prefetching* improved the performance of benchmark *410.bwaves* at the expense of the other three benchmarks. To understand the performance benefits provided by *Focused Prefetching*, we studied the number of prefetches generated and the number of useful prefetches for naive and *Focused Prefetching*. A prefetch is useful if the cache block brought in by it is accessed at the least once before being evicted from the cache. Figure 22(a) shows that similar to the single core scenario, lesser number of prefetches are generated by *Focused Prefetching*. Just as in the single-core scenario, *Focused Prefetching* generates highly accurate prefetches, as can be seen from Figure 22(b), which shows the increase in useful prefetches for *Focused Prefetching* compared to the naive baseline prefetching. While *Focused Prefetching*

10. We report geometric mean of the observed speedup.

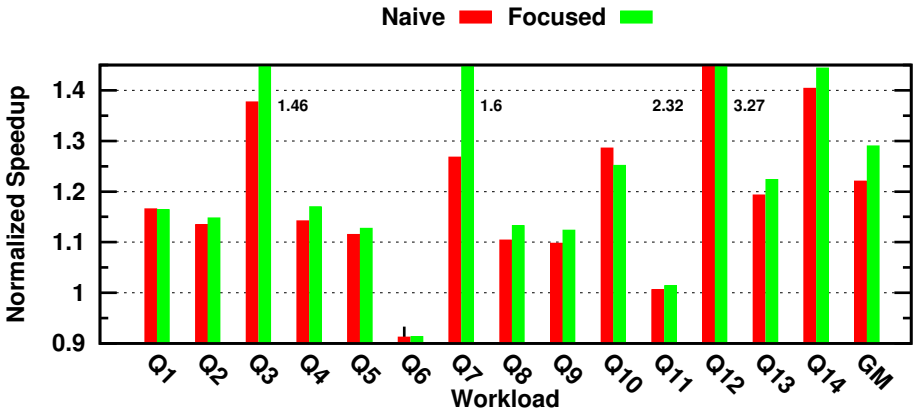
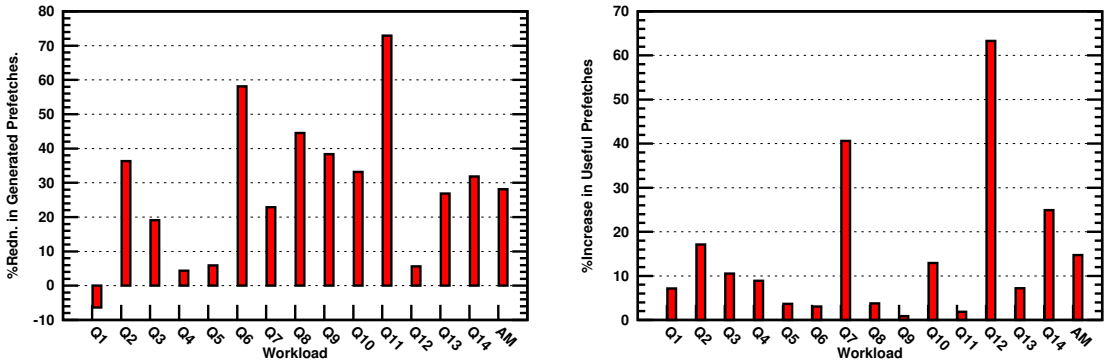


Figure 21: Quad-Core: Harmonic Mean Speedup of Baseline (Naive) Prefetching and Focused Prefetching Normalized to that of No Prefetching.



(a) Reduction in Prefetches Generated

(b) Increase in Useful Prefetches

Figure 22: Focused Prefetching Vs Baseline – While *Focused Prefetching* generates lesser number of prefetches compared to naive prefetching, it generates more accurate prefetches which is reflected in terms of the increase in the absolute number of useful prefetches experienced by *Focused Prefetching*.

generates 28% less prefetches compared to naive prefetching, it experiences 14.7% more prefetch hits compared to the baseline prefetching scenario. Workloads Q7 and Q12 show the maximum speedup in terms of harmonic speedup metric. It is interesting to note that while gain in the workload Q7 comes from a combination of eliminating wasted/useless prefetches (see Figure 22(a)) and additional hits due to more accurate prefetches(Figure 22(b)), workload Q12 gains mainly due to the additional hits(useful prefetches) generated by *Focused Prefetching*.

In essence, we show that *Focused Prefetching* scales seamlessly to multi-core scenario and can help improve the performance of prefetchers attached to shared caches.

6. Related Work

The works related to contributions made in this paper can be classified into three major categories: *tracking commit stalls, prefetching mechanisms* and *filtering prefetches*.

6.1. Tracking Commit Stalls

Tracking commit stalls experienced by a program and treating them as an indicator for the DRAM performance of the system has been carried out in [23]. *Scavenger* [24] makes an observation that loads missing in L2 account for a significant fraction of stall time, which is similar to ours. While [24], focuses on the misses from an address point of view and suggests cache structure reorganization, we focus on the stalls from an instruction point of view and focus on improving prefetching performance with out any modifications to existing prefetchers. Loads blocking the ROB often is also demonstrated in [25] and it proposes load speculation when a load stalls at commit. In short, they use commit stalls to filter the load speculation that needs to be carried out. But the major difference is the fact that they try to eliminate all the commit stalls rather than focusing on a few or the instructions that account for a lot of them as is done in this work.

6.2. Prefetching Mechanisms

Global History Buffer [8] has been shown to be the most effective way to track misses and also provides the flexibility to implement a variety of prefetching schemes on top of it. Earlier prefetching mechanisms used sequential [4] or next-line prefetching, while Markov predictors for prefetching proposed in [5] identified complex patterns in the miss stream. The popular prefetching scheme of tracking multiple streams in parallel, stream buffer is proposed in [6]. Latter research also showed that it is profitable to use either a stride [3] or Markov prefetcher [19] with stream buffers. Our approach is oblivious to the underlying prefetching mechanism used and helps by filtering the input stream seen by the prefetcher to improve the accuracy and efficiency.

6.3. Filtering Prefetches

Not treating all the loads as equal, and focusing only on a few of them was first proposed in [18]. A complex tracking mechanism and large prediction structures are used to identify and predict the criticality of a load in [18]. However their performance evaluation revealed poorer performance compared to naive prefetching at L2 and resulted in a loss in performance compared to no prefetching at L1. One of the criteria employed in [18] to identify critical loads is to measure the number of instructions issued in a certain number of cycles following the issue of a load. If the number of instructions issued is below a certain predetermined threshold, the load is classified as critical. The major problem with this approach of tracking at issue is the fact that the tracking needs to be carried out for multiple loads in parallel and if one of them is critical enough to affect the issue, all the other loads will get wrongly classified as critical.

In *Feedback Directed Prefetching (FDP)* [11], the filtering of the generated prefetches is carried out based on the accuracy, timeliness and pollution caused by the prefetcher. The filtering is achieved by controlling the prefetch degree of the prefetcher. The mechanism involved in throttling the prefetchers are reactive and are not aware of the relative impact on performance of the loads that suffer the misses. FDP filters prefetches once they are generated while we filter the training stream

seen by the prefetcher. This allows FDP to complement our scheme without any negative effects. More recent works use criterion similar to FDP to manage multiple prefetchers [26, 27].

A static filter which enables prefetching for a set of loads has been proposed in [28]. Unlike *Focused Prefetching*, it requires a profiling run and requires knowledge of the underlying prefetch mechanism. Also the filtering criteria is not performance oriented but is determined by the regular behaviour observed in the miss stream of a particular load, which might lead to an improvement in the accuracy of prefetches.

A filter based on usefulness of the prefetches is proposed in [29]. The scheme works by filtering prefetches once they are generated on a per prefetch basis. Like FDP, this scheme is also orthogonal to *Focused Prefetching*.

7. Conclusions

This paper proposed prefetching enhancements that target the prefetching efforts on a small set of loads incurring majority of commit stalls. This resulted in gains in performance and reduced memory traffic over naive prefetching. To summarize, the key contributions made in this paper are:

- We observe that close to 60% of the commit stalls are caused by loads and that a small set of loads, referred to as LIMCOS incur most of these stalls.
- We propose simple hardware structures called *Classifiers* which are entirely off the critical path to identify the occurrences of the LIMCOS loads.
- We demonstrate an application of the *Classifiers* to improve the performance gains from prefetching in *Focused Prefetching*. We show that focusing prefetching efforts on LIMCOS loads can lead to gains in performance, reduction in the memory traffic and improved prefetch accuracy.
- We demonstrate the performance benefits provided by *Focused Prefetching* in a multi-core scenario.
- We demonstrate another application of *Classifiers* in the form of filtering prefetches in global history based prefetchers.
- We also demonstrate that the criterion of commit stalls is better than other well known criteria like criticality [12] and delinquent loads [13].

References

[1] A. Cristal, D. Ortega, J. Llosa, and M. Valero, "Out-of-order commit processors," in *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, (Washington, DC, USA), p. 48, IEEE Computer Society, 2004.

[2] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, (New York, NY, USA), pp. 176–186, ACM, 1991.

[3] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic, "Memory-system design considerations for dynamically-scheduled processors," in *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 133–143, ACM, 1997.

[4] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," *SIGMICRO Newsl.*, vol. 23, no. 1-2, pp. 102–110, 1992.

[5] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 252–263, ACM, 1997.

[6] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, (New York, NY, USA), pp. 364–373, ACM, 1990.

[7] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith, "Ac/dc: An adaptive data cache prefetcher," in *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, (Washington, DC, USA), pp. 135–145, IEEE Computer Society, 2004.

[8] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, (Washington, DC, USA), p. 96, IEEE Computer Society, 2004.

[9] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems, "Guided region prefetching: a cooperative hardware/software approach," in *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 388–398, ACM, 2003.

[10] R. Cooksey, S. Jourdan, and D. Grunwald, "A stateless, content-directed data prefetching mechanism," in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, (New York, NY, USA), pp. 279–290, ACM, 2002.

[11] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, (Washington, DC, USA), pp. 63–74, IEEE Computer Society, 2007.

[12] B. Fields, S. Rubin, and R. Bodík, "Focusing processor policies via critical-path prediction," *SIGARCH Comput. Archit. News*, vol. 29, no. 2, pp. 74–85, 2001.

[13] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: long-range prefetching of delinquent loads," in *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 14–25, ACM, 2001.

[14] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for mlp-aware cache replacement," in *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, (Washington, DC, USA), pp. 167–178, IEEE Computer Society, 2006.

[15] R. D. Doug, D. Burger, S. W. Keckler, and T. Austin, "Sim-alpha: a validated, execution-driven alpha 21264 simulator," tech. rep., 2001.

[16] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, (Los Alamitos, CA, USA), pp. 81–87, IEEE Computer Society Press, 1981.

[17] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 45–57, ACM, 2002.

[18] R. D.-c. Ju, A. R. Lebeck, and C. Wilkerson, "Locality vs. criticality," in *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture* (S. T. Srinivasan, ed.), (New York, NY, USA), pp. 132–143, ACM, 2001.

[19] T. Sherwood, S. Sair, and B. Calder, "Predictor-directed stream buffers," in *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, (New York, NY, USA), pp. 42–53, ACM, 2000.

[20] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The m5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, pp. 52–60, 2006.

[21] I. Ganusov and M. Burtscher, "Future execution: A prefetching mechanism that uses multiple cores to speed up single threads," *ACM Trans. Archit. Code Optim.*, vol. 3, pp. 424–449, December 2006.

[22] K. Luo, J. Gummaraju, and M. Franklin, "Balancing thoughput and fairness in smt processors," pp. 164 –171, 2001.

[23] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 146–160, IEEE Computer Society, 2007.

[24] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez, "Scavenger: A new last level cache architecture with global block priority," in *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 421–432, IEEE Computer Society, 2007.

[25] N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martinez, "Checkpointed early load retirement," in *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, (Washington, DC, USA), pp. 16–27, IEEE Computer Society, 2005.

[26] E. Ebrahimi, O. Mutlu, and Y. Patt, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pp. 7 –17, 2009.

[27] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 316–326, ACM, 2009.

[28] V. Srinivasan, "A static filter for reducing prefetch traffic," tech. rep., 1999.

[29] X. Zhuang and H.-H. Lee, "A hardware-based cache pollution filtering mechanism for aggressive prefetches," pp. 286 –293, oct. 2003.