Focused Prefetching: Performance Oriented Prefetching Based On Commit Stalls

R. Manikantan† and R. Govindarajan†‡
†Dept. of Computer Science & Automation
‡Supercomputer Education & Research Center
Indian Institute of Science, Bangalore,India
{rmani,govind}@csa.iisc.ernet.in

Abstract

Loads that miss in L1 or L2 caches and 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.*

We study an application of these classifiers to prefetching. 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.

Categories and Subject Descriptors: C.1 [Processor Architectures]
General Terms: Design, Experimentation, Performance.

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 [4]. 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 inspite 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.

A wide variety of prefetchers have been studied for data caches [1, 6, 8, 9, 10, 15, 16, 23]. All these prefetchers are normally trained on the miss/access stream and identify useful patterns/trends among the accesses. The identified trend is used to predict the addresses that are most likely to be accessed in the near future. 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 [20].

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 . We refer to these loads as LIMCOS (Loads Incurring Majority of COmmit Stalls). Simple classifiers based on history allow us to easily identify this small set of loads that stall the pipeline frequently. The classifiers are off the critical path and work by tracking the stalls experienced by individual loads.

Focused Prefetching uses the classifiers to filter the training stream seen by the prefetcher, i.e., only the misses suffered by loads identified by the classifier act as the training stream for the prefetcher. By focusing on misses suffered by LIMCOS, it allows the prefetcher to eliminate 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 [16]. Also this gain in performance is

[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.

ICS'08, June 7-12, 2008, Island of Kos, Aegean Sea, Greece.

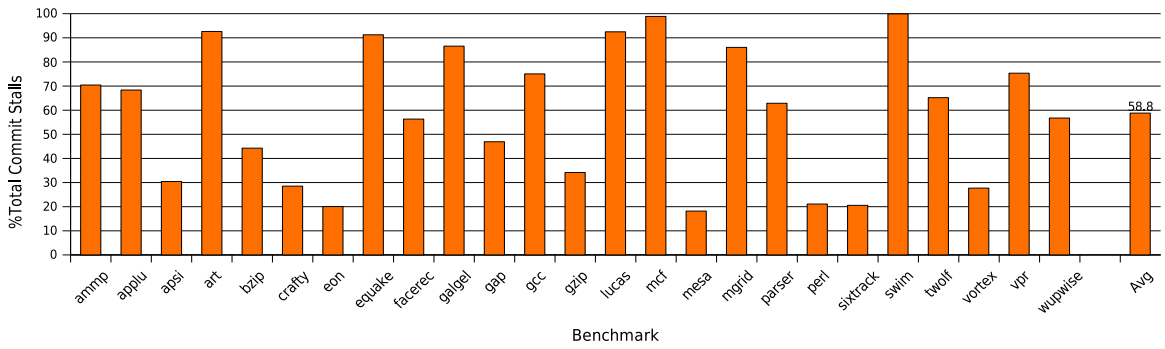


Figure 1: Fraction of commit stall cycles that can be attributed to loads

achieved along with a 20% reduction in memory traffic and a 61% improvement in the accuracy of prefetches. Finally, we demonstrate that the classification criterion of load commit stalls is better than existing criteria like either criticality [7] or load-specific criteria like delinquent loads [3]. 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 [16] 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. On analysing it further by looking at the contribution of individual loads, led us to observe a few interesting trends. 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.

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 [3], 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 [3] 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 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

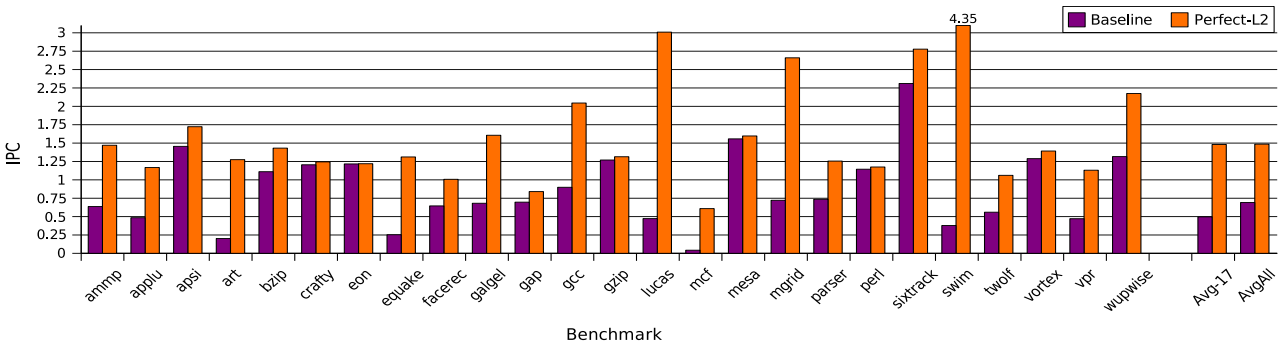


Figure 2: IPC for baseline (no prefetch) and perfectL2

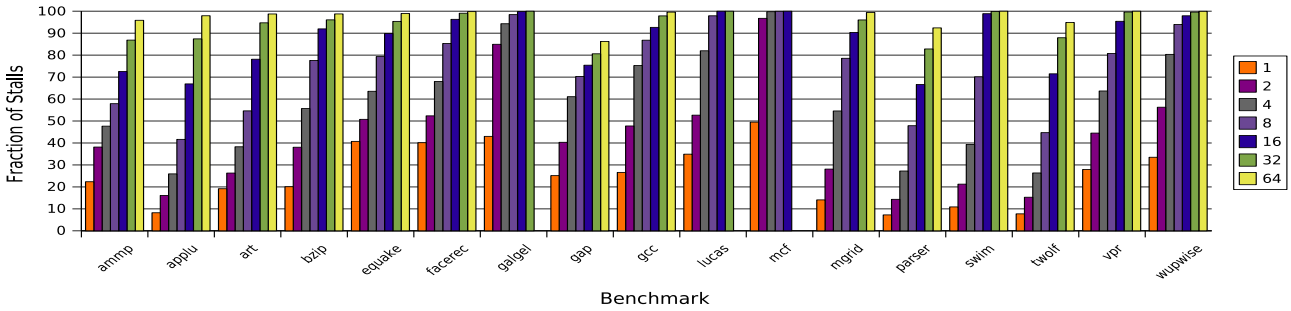


Figure 3: Individual loads and their contribution to commit stalls: Fraction of commit stalls accounted for by various number of static loads

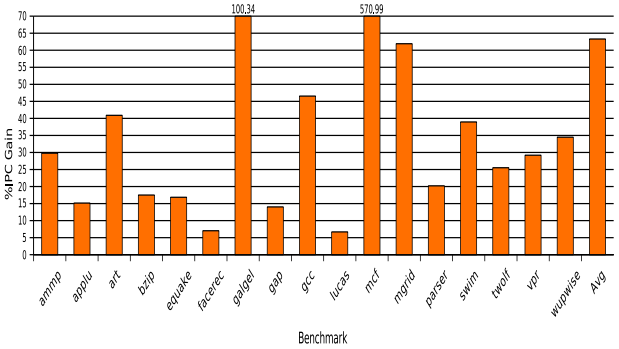


Figure 4: Gain in performance for *Instant Replacement* over baseline

can be attributed to the effects of Memory Level Parallelism (MLP) [17]. *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.

3. 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, it is obvious 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. The classifiers work by tracking the commit stalls experienced by each load and make their decision as to whether the loads experience frequent commit stalls.

We study 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.

3.1 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 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, 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.

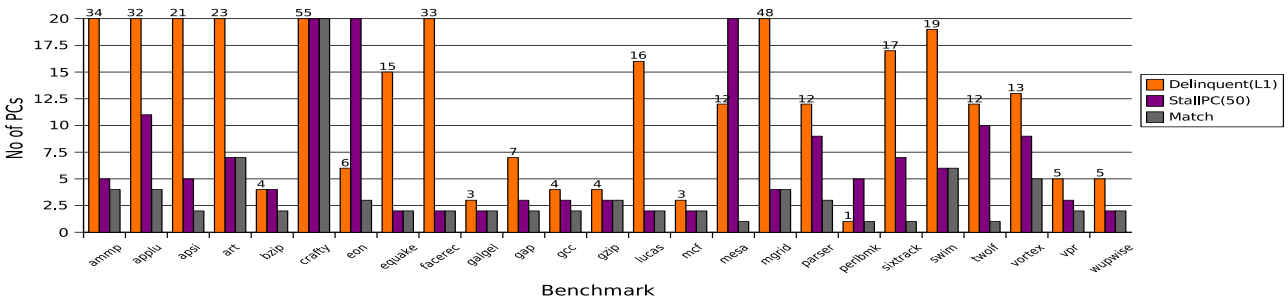


Figure 5: Relation to *Delinquent Loads*: Overlap between loads accounting for 50% of stalls and 50% of misses

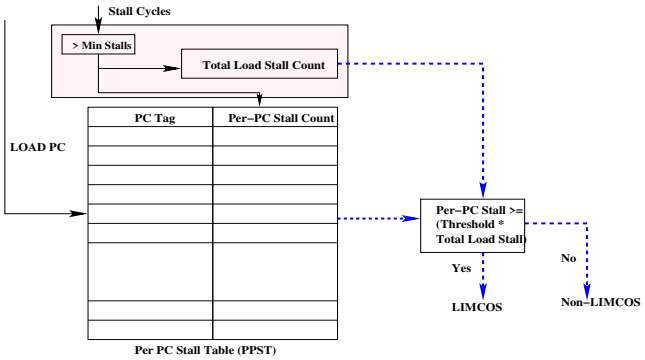


Figure 6: Counting classifier organization

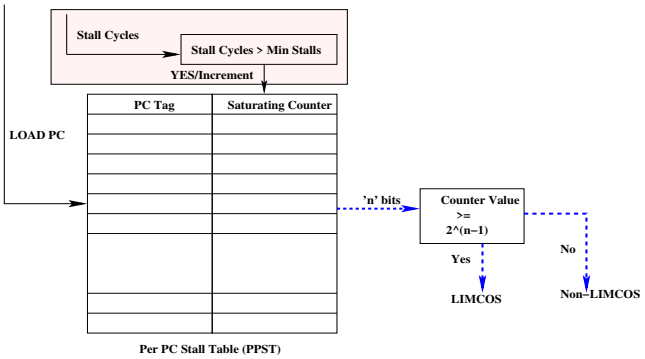


Figure 7: Confidence based classifier organization

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 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.

3.2 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 7. 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 7, 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. 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 the confidence based classifier will classify the load as frequently stalling (as there is no decrement of confidence) and will enable focused prefetching. The counting classifier, on the

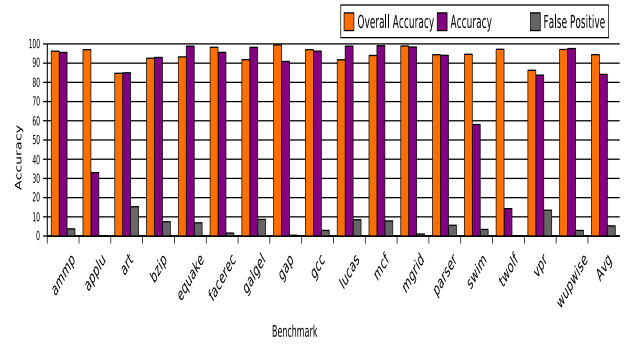


Figure 8: Accuracy of counting classifier: 32 Entries, Min Stalls 16 cycles, Threshold 1/32

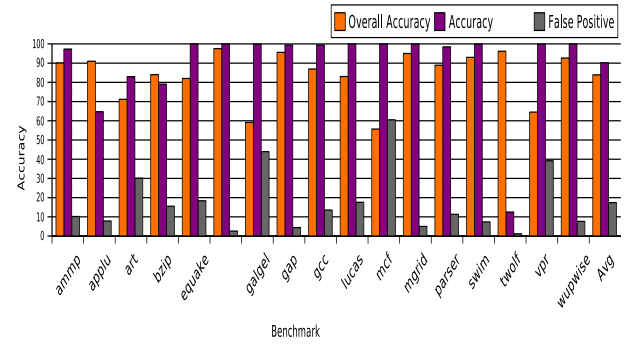


Figure 9: Accuracy of confidence classifier: 32 Entries, 8 way associative, Min Stall 32 cycles

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.

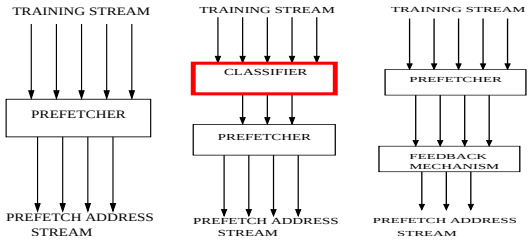
3.3 Evaluating the Classifiers

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 8 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 9 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 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 lack of a decrement of 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 inspite of using only 32 entries.

[3]The trends observed were similar with higher coverage like LIMCOS-80.



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

Figure 10: Focused Prefetching and other prefetch mechanisms

4. FOCUSED PREFETCHING

Focused prefetching is an application of the classifiers mentioned in Section 3. Focused prefetching is a filtering mechanism that helps any prefetcher to focus more on misses that have a bigger impact on performance.

Any existing prefetcher, as shown in Figure 10(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 10(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 [16], 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)* [20]. FDP, as shown in Figure 10(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 [5]. The machine model and other relevant parameters are presented in Table 1. Each level of cache has 32 MSHRS [12] out of which 16 are reserved for prefetches. Regular accesses are given priority over prefetches. We used the early single simulation point [18] 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) [16]. 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 adapted as it is shown to be one of the best performing prefetch algorithm in [16]. 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 classifiers used are the ones evaluated in Section 3.

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	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
Memory Latency	Minimum 225 cycles
Prefetcher	At L2 - 512 Entry 16 Index GHB Per PC Delta Correlation 512 Entry 256 Index was also evaluated for Baseline
Prefetch Degree	8
Counting Classifier	32 Entries, Lower Limit 16 and Threshold 1/32
Confidence Classifier	32 Entries, 8 Way Associative, Lower Limit 32

Table 1: Machine parameters

5.2 Performance and Traffic Gains with Focused Prefetching

By using the classifiers to focus only on LIMCOS loads, we expect *Focused Prefetching* to eliminate most of the commit stalls encountered and hence have a positive impact on performance. 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 11 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, *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

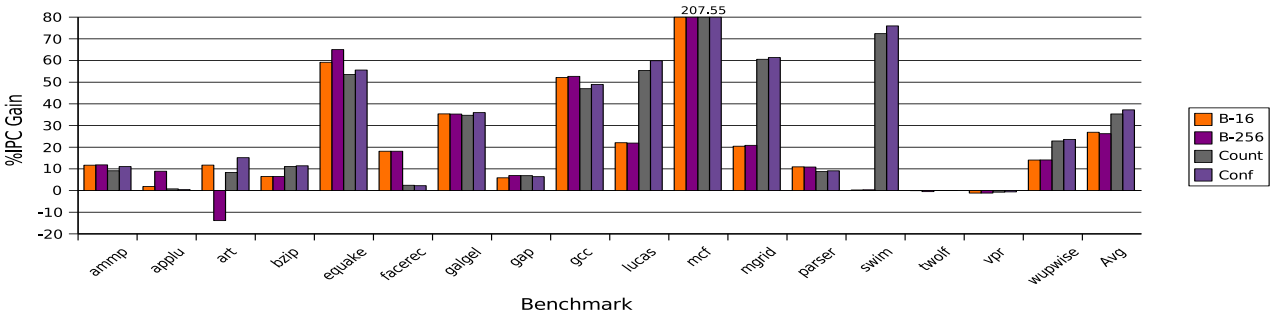


Figure 11: Performance gains of GHB with PC-Delta Correlation and Focused Prefetching over a baseline with no prefetcher

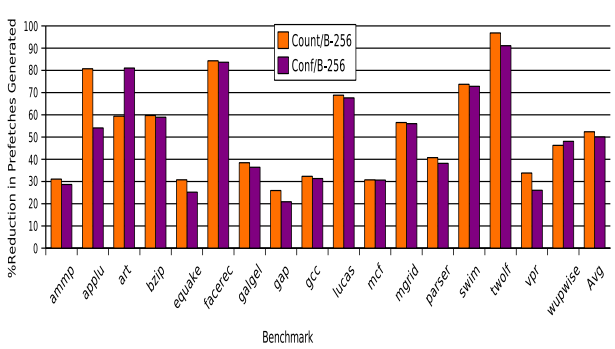


Figure 12: Reduction in the number of prefetches generated by Focused Prefetching

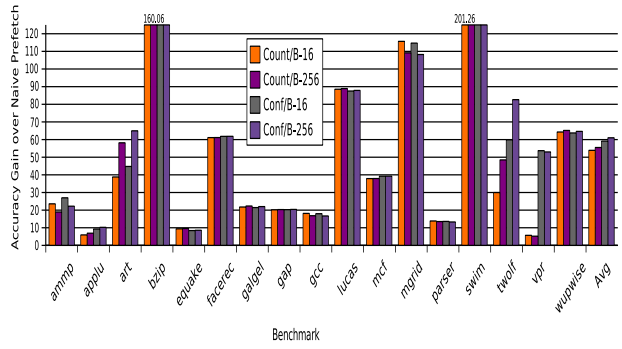


Figure 13: Improvement in prefetch accuracy due to Focused Prefetching

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[4] is brought down by 75%. This results in a drop in the performance compared to naive prefetching.

Figure 12 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 [20].

The gains in accuracy over B-16 and B-256 for *Counting Classifier* and *Confidence Based Classifier* are shown in Figure 13. *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 14 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* enables one to eliminate the misses that matter and achieves more performance by virtue of more relevant prefetches.

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 15. 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. For the entire set of 25 benchmarks, the mem-

[4]prefetches servicing a demand access.

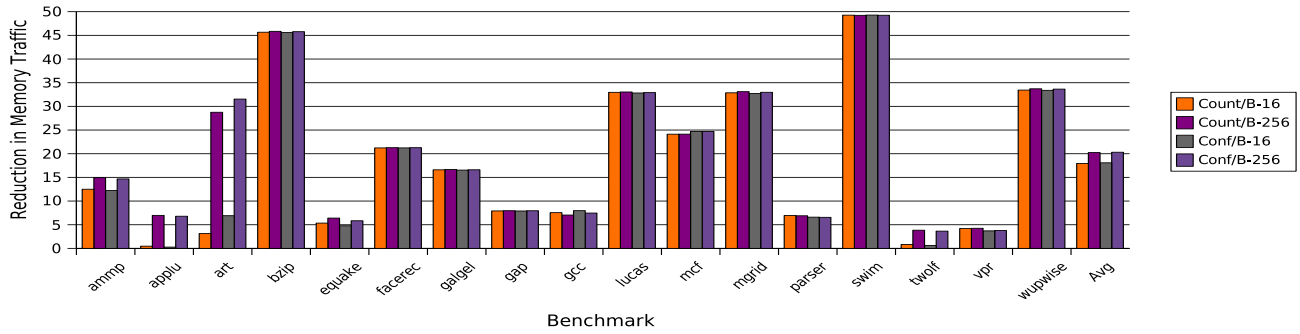


Figure 14: Reduction in memory traffic by employing Focused Prefetching

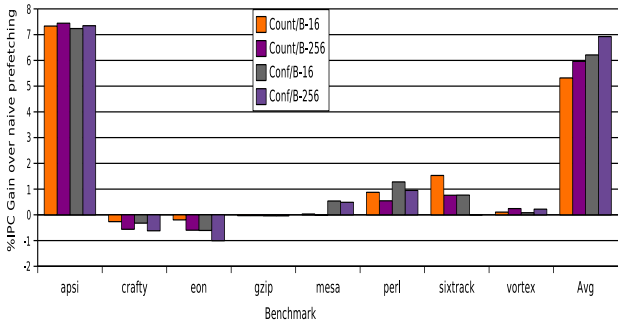


Figure 15: Gain in performance with Focused Prefetching for benchmarks not discussed in detail

ory 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 Relation to Other Criteria

In this section, we present quantitative comparisons with two of the most closely related criteria to commit stalls viz., *criticality* [7] and *delinquent loads* [3].

5.3.1 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 [21]. 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 [7]. The methodology proposed in [7] 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 [7] to identify a set of static loads that account for most of the critical misses suffered at L2 cache.

5.3.2 Delinquent Loads

Section 2 showed that there is a partial overlap between delinquent loads [3] and LIMCOS.

5.3.3 Performance Comparison

As critical loads are identified accurately using an offline 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 [18]. 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. 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 that account for most of the misses. 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 across the three criteria. LIMCOS-50 is used as the basic criteria 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, GHB that can track 512 misses from 16 different PCs (16 Index table entries) and using a perPC Delta Correlation to generate the prefetch addresses [16].

Figure 16 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

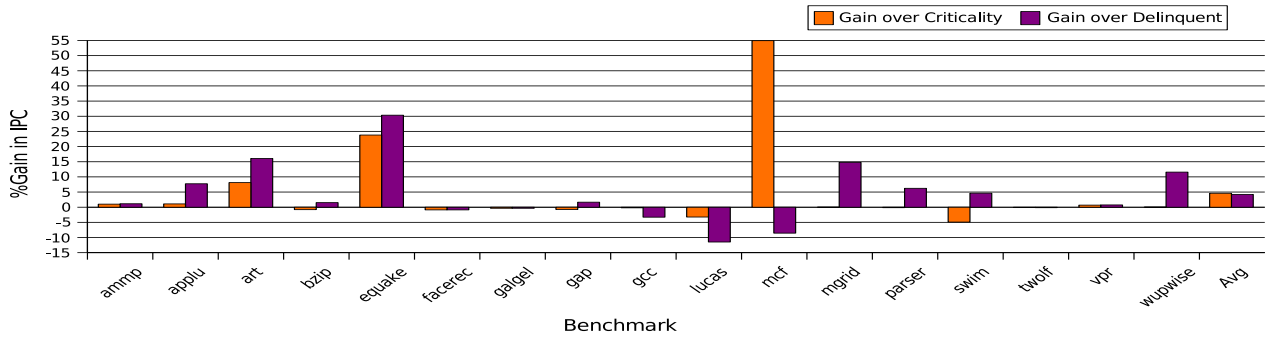


Figure 16: Performance gains of Focused Prefetching over Criticality and Delinquent Loads

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.

5.4 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 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 [10] 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 [6] or Markov prefetcher [19] along with it. The delta correlation predictor used earlier can be thought of as an approximation of the Markov predictor [9]. 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 17. For the set of 25 SPEC benchmarks[5], 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%.

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.*

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

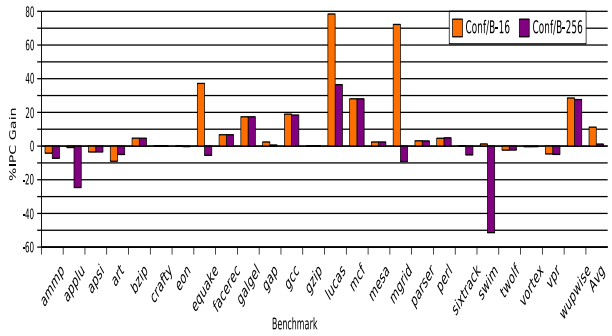


Figure 17: Performance gains provided by Focused Prefetching at L1

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 [14]. *Scavenger* [2] makes an observation that loads missing in L2 account for a significant fraction of stall time, which is similar to ours. While [2], 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 [11] 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 [16] 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 [8] or nextline prefetching, while Markov predictors for prefetching proposed in [9] identified complex patterns in the miss stream. The popular prefetching scheme of tracking multiple streams in parallel, stream buffer is proposed in [10]. Latter research also showed that it is profitable to use either a

stride [6] 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 [21]. A complex tracking mechanism and large prediction structures are used to identify and predict the criticality of a load in [21]. 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 [21] 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)* [20], the filtering of the generated prefetches is carried out based on the accuracy, timeliness and pollution caused by the prefetcher. FDP is reactive and is 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.

A static filter which enables prefetching for a set of loads has been proposed in [22]. 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 [24]. 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 *Focused Prefetching*, a scheme that focuses the prefetching efforts on a small set of loads incurring majority of commit stalls. 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* that 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 also demonstrate that the criterion of commit stalls is better than other well known criteria like criticality [7] and delinquent loads [3].

8. REFERENCES

[1] J. Baer and T. Chen, An effective on-chip preloading scheme to reduce data access penalty. In *Proc. of Supercomputing'91, 1991.*

[2] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, J.F. Martinez, Scavenger: A New Last Level Cache Architecture With Global Block Priority. In *Proc. of Int. Symp. on Microarchitecture,* 2007.

[3] 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 *Proc. of Int. Symp. Computer Architecture-28,* 2001.

[4] A. Cristal, D. Ortega, J. Llosa and M. Valero, Out-of-order commit processors. In *Proc. of Int. Symp. on High Performance Computer Architecture,* 2004.

[5] R.Desikan, D.C. Burger, S.W. Keckler and T. Austin, Sim-alpha: a Validated, Execution-Driven Alpha 21264 Simulator. *The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-01-23,2001.*

[6] K. Farkas, P. Chow, N. Jouppi and Z. Vranesic, Memory-system design considerations for dynamically-scheduled processors. In *Proc. of Int. Symp. Computer Architecture,* 1997.

[7] B. Fields, S. Rubin and R. Bodik, Focusing processor policies via critical-path prediction. In *Proc. of Int. Symp. Computer Architecture,* 2001.

[8] J.W.C. Fu and J.H. Patel, Stride directed prefetching in scalar processors. In *Proc. of Int. Symp. on Microarchitecture,* 1992.

[9] D. Joseph and D. Grunwald, Prefetching Using Markov Predictors. In *IEEE Trans. on Computer Systems,* 1999.

[10] N.P. Jouppi, Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. of Int. Symp. Computer Architecture,* 1990.

[11] N. Kirman, M. Kirman, M. Chaudhuri and J.F. Martinez, Checkpointed Early Load Retirement. In *Proc. of Int. Symp. on High Performance Computer Architecture,* 2005.

[12] D. Kroft, Lockup-free instruction fetch/prefetch cache organization. In *Proc. of Int. Symp. Computer Architecture,* 1981.

[13] W.F. Lin, S.K. Reinhardt, D. Burger and T.R. Puzak, Filtering superfluous prefetches using density vectors. In *Proc. of Int. Conf. on Computer Design,* 2001.

[14] O. Mutlu and T. Moscibroda, Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proc. of Int. Symp. on Microarchitecture,* 2007.

[15] K.J. Nesbit, A.S. Dhodapkar and J.E. Smith, AC/DC: An adaptive data cache prefetcher. In *Proc. of Int. Conf. on Parallel Architectures and Compilation Techniques*, 2004.

[16] K.J. Nesbit and J.E. Smith, Data Cache Prefetching Using a Global History Buffer. In *Proc. of Int. Symp. on High Performance Computer Architecture,* 2004.

[17] M.K. Qureshi, D.N. Lynch, O. Mutlu, Y.N. Patt, A Case for MLP-Aware Cache Replacement. In *Proc. of Int. Symp. Computer Architecture,* 2006.

[18] T. Sherwood, E. Perelman, G. Hamerly and B. Calder, Automatically Characterizing Large Scale Program Behaviour. In *Proc. of Int. Conf. on Architectural Support for Programming Languages and Operating Systems,* 2002.

[19] T. Sherwood, S. Sair and B. Calder, Predictor-Directed Stream Buffers. In *Proc. of Int. Symp. on Microarchitecture,* 2000.

[20] S. Srinath, O. Mutlu, H. Kim, Y.N. Patt, Feedback Directed Prefetching: Improving the Performance and Bandwidth Efficiency of Hardware Prefetchers. In *Proc. of Int. Symp. on High Performance Computer Architecture,* 2007.

[21] S.T. Srinivasan, R.D-C. Ju, A.R. Lebeck, C.R. Wilkerson, Locality vs. Criticality. In *In Proc. of Int. Symp. Computer Architecture,* 2001.

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

[23] Z. Wang, D. Burger, K. McKinley, S. Reinhardt and C. Weems, Guided Region Prefetching: A Cooperative Hardware/Software Approach. In *Proc. of Int. Symp. Computer Architecture,* 2003.

[24] X. Zhuang and H.H.S. Lee, A hardware-based cache pollution filtering mechanism for aggressive prefetches. In *Proc. of Int. Conf. on Parallel Processing,* 2003.