Band-Pass Prefetching: An Effective Prefetch Management Mechanism Using Prefetch-Fraction Metric in Multi-Core Systems

ASWINKUMAR SRIDHARAN, BISWABANDAN PANDA, and ANDRE SEZNEC,
INRIA Rennes, France

In multi-core systems, an application's prefetcher can interfere with the memory requests of other applications using the shared resources, such as last level cache and memory bandwidth. In order to minimize prefetcher-caused interference, prior mechanisms have been proposed to dynamically control prefetcher aggressiveness at runtime. These mechanisms use several parameters to capture prefetch usefulness as well as prefetcher-caused interference, performing aggressive control decisions. However, these mechanisms do not capture the actual interference at the shared resources and most often lead to incorrect aggressiveness control decisions. Therefore, prior works leave scope for performance improvement.

Toward this end, we propose a solution to manage prefetching in multicore systems. In particular, we make two fundamental observations: First, a positive correlation exists between the accuracy of a prefetcher and the amount of prefetch requests it generates relative to an application's total (demand and prefetch) requests. Second, a strong positive correlation exists between the ratio of total prefetch to demand requests and the ratio of average last level cache miss service times of demand to prefetch requests. In this article, we propose Band-pass prefetching that builds on those two observations, a simple and low-overhead mechanism to effectively manage prefetchers in multicore systems. Our solution consists of local and global prefetcher aggressiveness control components, which altogether, control the flow of prefetch requests between a range of prefetch to demand requests ratios. From our experiments on 16-core multi-programmed workloads, on systems using stream prefetching, we observe that Band-pass prefetching achieves 12.4% (geometric-mean) improvement on harmonic speedup over the baseline that implements no prefetching, while aggressive prefetching without prefetcher aggressiveness control and state-of-the-art HPAC, P-FST, and CAFFEINE achieve 8.2%, 8.4%, 1.4%, and 9.7%, respectively. Further evaluation of the proposed Band-pass prefetching mechanism on systems using AMPM prefetcher shows similar performance trends. For a 16-core system, Band-pass prefetching requires only a modest hardware cost of 239 bytes.

CCS Concepts: • **Computer systems organization → Parallel architectures**; **Processors and memory architectures**

Additional Key Words and Phrases: Multicore, memory systems, intercore interference, prefetching

ACM Reference Format:
Aswinkumar Sridharan, Biswabandan Panda, and Andre Seznec. 2017. Band-pass prefetching: An effective prefetch management mechanism using prefetch-fraction metric in multi-core systems. ACM Trans. Archit. Code Optim. 14, 2, Article 19 (June 2017), 27 pages.
DOI: http://dx.doi.org/10.1145/3090635

This work is supported by ERC Advanced Grant DAL No. 267175 and partially by an Intel research grant.
Authors' addresses: A. Sridharan, B. Panda, and A. Seznec, Campus de Beaulieu, INRIA Rennes, 35042 Rennes Cedex; emails: {aswinsridharan, biswa.uce}@gmail.com, andre.seznec@inria.fr.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2017 ACM 1544-3566/2017/06-ART19 $15.00
DOI: http://dx.doi.org/10.1145/3090635

1. INTRODUCTION

An aggressive hardware prefetcher may completely hide the latency of off-chip memory accesses. However, it may cause severe interference at the shared resources (last level cache and memory bandwidth) of a multi-core system [Ebrahimi et al. 2009, 2011; Wu et al. 2011; Seshadri et al. 2015; Panda and Balachandran 2015; Jimenez et al. 2015; Panda 2016; Lee et al. 2008; Liu and Solihin 2011; Bitirgen et al. 2008]. To manage prefetching in multi-core systems, prior studies [Srinath et al. 2007; Ebrahimi et al. 2009, 2011; Panda and Balachandran 2015; Panda 2016] have been proposed to dynamically control (also known as throttling) the prefetcher aggressiveness by adjusting the prefetcher-configuration at runtime. These mechanisms make dynamic throttling decisions by computing several parameters, such as prefetch-accuracy, lateness, prefetcher-caused interference at the last level cache and Dynamic Random-Access Memory (DRAM) in the form of pollution, row-buffer, bus, and bank conflicts.

Problem: Prior works such as Hierarchical Prefetcher Aggressiveness Control (HPAC) [Ebrahimi et al. 2009] and CAFFEINE [Panda and Balachandran 2015] do not completely alleviate the problem of prefetcher-caused interference in multi-core systems. With HPAC, we observe that the use of multiple metrics (driven by their threshold values) does not capture the actual interference in the system, and in *most cases* leads to incorrect throttling decisions. With CAFFEINE, approximate estimation of the average last level cache miss penalty leads to *biased* throttling decisions, *overlooking* interference caused due to prefetchers. Accordingly, such prior works provide scope for further performance improvements.

Our Solution: Toward proposing a solution to manage interference caused by prefetchers in multi-core systems, we make two fundamental observations. First, for a given application, *the fewer the number of prefetch requests generated, the less likely that they are useful*. That is, a strong positive correlation exists between the accuracy of a prefetcher and the amount of prefetch requests generated for an application relative to its total prefetch and demand requests. Second, *the more the aggregate number of prefetch requests in the system, the higher the miss penalty on demand misses at the last level cache*. In particular, we observe a strong positive correlation between the ratio of average miss service times of demand to prefetch misses and the ratio of aggregate prefetch (misses)[1] to demand misses at the shared LLC-DRAM interface.

Based on these two observations, we introduce *prefetch-fraction* metric that *infers* the (i) usefulness (in terms of prefetch-accuracy) of prefetching to an application and (ii) interference caused by a prefetcher at the shared LLC-DRAM interface. We define prefetch-fraction of an application as the fraction of L2 prefetch requests (the prefetcher generates) with respect to the total requests (demand misses, L1 and L2 prefetch requests). To infer the usefulness of prefetching to an application, we compute prefetch-fraction for each application independently at the private L2-LLC interfaces. To infer interference due to a prefetcher, we compute prefetch-fraction for each application at the shared LLC-DRAM interface.

Notion of Band-pass Prefetching: Based on the inference drawn from the computed prefetch-fraction values, our proposed mechanism applies simple prefetcher aggressiveness control at two levels. First, at the private L2 (application) level when the inferred prefetch-accuracy is low. Second, at the shared LLC-DRAM interface (globally), when prefetch requests are likely to delay demand misses. The two mechanisms independently control the flow of prefetch requests between a range (band) of

[1]By prefetch misses, we refer to the L2 prefetch requests generated by the prefetcher sitting beside each private L2 cache that miss and leave LLC for DRAM access.

prefetch-to-demand ratios. This is analogous to Band-pass filtering in signal processing systems [Oppenheim et al. 1996]. A band-pass filter consists of high-pass and low-pass components: high-pass allows signal frequencies that are only higher than a threshold value, while low-pass allows only signal frequencies that are lower than a threshold value. Together, the two filters allow only a band of signal frequencies to pass through. Similarly, our two mechanisms allow only prefetch requests that are between a range of prefetch-to-demand ratios to flow through from LLC to DRAM. Hence, we refer to our solution as Band-pass prefetch filtering or simply, Band-pass prefetching. Precisely, we make the following contributions:

—We identify key issues with the state-of-the-art prefetcher aggressiveness control mechanisms: HPAC [Ebrahimi et al. 2009] and CAFFEINE [Panda and Balachandran 2015] (Section 2).
—We make two fundamental observations: (i) prefetch-accuracy strongly correlates with the ratio of number of L2 prefetch requests generated to an application's total requests (demands and prefetches from L1 and L2 caches) and (ii) prefetcher-caused interference (delay on demand misses) strongly correlates with the ratio of total prefetch requests to the total demands in the system (Section 3).
—We introduce band-pass prefetching (Section 4), a simple mechanism that measures the fraction of prefetch requests with respect to demands at L2-LLC and LLC-DRAM interfaces, and controls the flow of prefetch requests between a range of prefetch to demand ratios. In terms of hardware requirement, for a 16-core system, our proposed mechanism contributes to less than 240 bytes (15 bytes per application) of hardware overhead in total.

2. BACKGROUND

This section provides a background on our baseline system and the definitions that we use throughout the article. We then briefly describe HPAC [Ebrahimi et al. 2009] and CAFFEINE [Panda and Balachandran 2015], two state-of-the-art prefetcher aggressiveness control mechanisms.

2.1. Baseline Assumptions and Definitions

In this article, our goal is to propose a mechanism that can manage prefetcher-caused interference in multi-core systems, and not a new prefetching mechanism itself. Throughout the article, we consider a system with a three level cache-hierarchy with private L1 and L2 caches. The Last Level Cache (LLC) is shared by all the cores. L1 caches feature a next-line prefetcher, while L2 features a *stream prefetcher, which we intend to control*. Our stream prefetcher model is closer to the implementations of Feedback Directed Prefetching (FDP) [Srinath et al. 2007] and IBM Power series of processors [Sinharoy et al. 2011]. It sits next to the L2 cache and gets trained by L2 misses and L2 prefetch-hits. Only one stream entry (a unique prefetchable context) is allocated per 4KB page. It tracks 32 outstanding streams and issues prefetch requests with prefetch-distance of 8 and prefetch-degree of 4.

Definitions. Throughout the article, we use the following terminologies: Prefetch-distance: The number of cache lines ahead of X that the prefetcher tries to prefetch, where X is the cache block address of the cache miss that allocated the current stream. Prefetch-degree: The number of prefetch requests issued when there is an opportunity to prefetch. Throttling up/down: A prefetcher's aggressiveness is defined in terms of its prefetch-distance and degree. Throttling up/down refers to increasing/decreasing the values of prefetch-distance and degree to control aggressiveness.

2.2. State-of-the-art

HPAC: HPAC consists of a per-core local and a shared global feedback component. While HPAC's local component (FDP [Srinath et al. 2007]) attempts to maximize the benefit of prefetching for an application, the global component attempts to minimize the interference caused by a prefetcher. The local component computes prefetch-accuracy, lateness, and pollution metrics local to an application. The global component computes interference related parameters, such as bandwidth consumed (in cycles) by prefetch requests, amount of time (bandwidth needed in cycles) demands of an application wait due to prefetch requests for a memory resource (BWN), and prefetcher-caused cache pollution (POL). For each application, HPAC also computes BWNO metric, which is the bandwidth requirement of other cores. HPAC assumes that BWNO of a prefetcher becomes high when its prefetch requests consume high bandwidth (BWC), and forces memory requests of other applications to wait. Based on the threshold values of these metrics, HPAC's global component *infers* an application to be *interfering-with-others* or *not*. If a prefetcher is found to be interfering, HPAC's global control throttles it down. Otherwise, it allows the decision of its local component (FDP).

Problem with HPAC: The issue with HPAC is its use of multiple metrics and the inference drawn from them. A given value of a metric does not reflect the runtime behavior of an application due to interference caused when large number of applications run on the system. For example, a prefetcher's accuracy drops down when its prefetch requests are delayed at the shared resources by the co-running applications. Similarly, HPAC uses Bandwidth Needed by Others (BWNO) parameter to account for the bandwidth requirement of all other applications in the system, except the one under consideration. When large applications run on a system, BWNO of an application tends to be higher, while its prefetcher may not consume much bandwidth (BWC). Under the scenario in which an application's prefetch-accuracy is low and BWNO parameter is high, HPAC infers the application to be *interfering-with-others* and decides to throttle-down its prefetcher. In contrast, an application with high-accuracy is throttled-up although its prefetch requests consume high bandwidth. We observe several instances of such scenarios where HPAC does not capture interference and makes incorrect throttling decisions.

CAFFEINE: CAFFEINE takes a fine-grained account of interference caused by prefetch requests at each of the shared resources, such as DRAM bus, banks, row-buffers, and shared last level cache. It accounts for the benefit of prefetching to an application by estimating the amount of cycles saved in terms of its off-chip memory accesses. It normalizes both interference and prefetch usefulness to a common scale of processor cycles, which it refers to as a prefetcher's net-utility. It uses both system-wide and per-core net-utilities to make throttling decisions. In particular, CAFFEINE throttles-up prefetcher when the system-wide net-utility is positive and throttles them down otherwise.

Problem with CAFFEINE: CAFFEINE estimates the average last level cache miss-penalty by accumulating the latency of individual memory requests (difference in arrival and start times) and then, computing the arithmetic-mean on this accumulated sum of latencies over all requests. The resulting mean value is approximated as the average miss-penalty. In doing so, CAFFEINE treats each memory request as an isolated event and does not take into account overlapping memory accesses inherent in applications. Therefore, when miss-penalty, which is *overestimated*, is used in its utility model, CAFFEINE overestimates the cycles saved on off-chip memory accesses due to prefetching. Hence, CAFFEINE's throttling decisions favor aggressive prefetching.

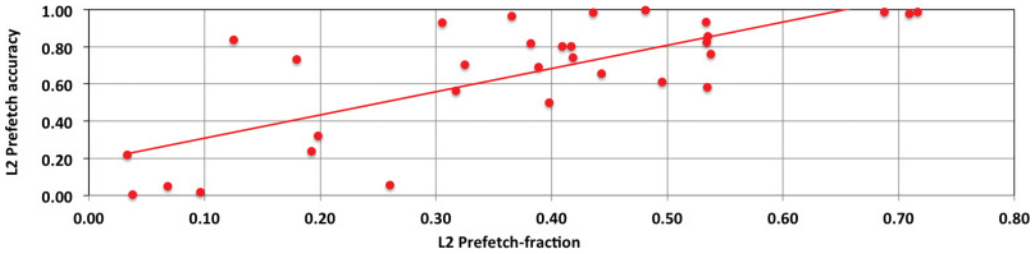

Fig. 1. Scatter plot showing positive correlation between L2 Prefetch-fraction versus L2 Prefetch-accuracy for benchmarks under the baseline aggressive stream prefetching: Pearson correlation coefficient: 0.76 and Spearman rank correlation: 0.68.

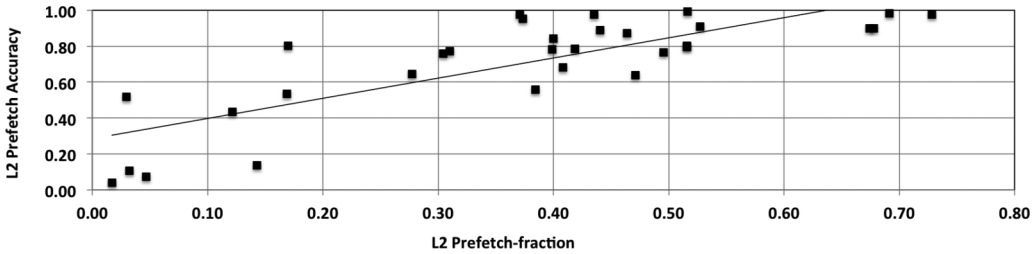

Fig. 2. Scatter plot showing positive correlation between L2 Prefetch-fraction versus L2 Prefetch-accuracy for benchmarks under Feedback directed prefetching: Pearson correlation coefficient: 0.80 and Spearman rank correlation: 0.75.

3. MOTIVATIONAL OBSERVATIONS

In this section, we discuss how Prefetch-fraction statistically captures both the usefulness (prefetch-accuracy) of prefetching and prefetch-caused interference (delay induced on demands by prefetch requests) at the shared memory bandwidth.

3.1. Correlation between Prefetch-accuracy and Prefetch-fraction

The amount of prefetch requests generated by a prefetcher depends on an application's access pattern and the ability of the prefetcher to capture it. If the pattern is not conceivable to the prefetcher, it generates fewer in-accurate prefetch requests. However, if the pattern is conceivable, and the application exhibits many prefetch-able contexts, the prefetcher generates a large number of accurate prefetch requests. In particular, usefulness of prefetching (in terms of prefetch-accuracy) depends on the fraction of L2 prefetch requests generated with respect to an application's total requests. Figures 1 and 2 illustrate the correlation between L2 prefetch-fraction and L2 prefetch-accuracy for applications (refer to Table III) for the baseline aggressive stream prefetcher and FDP, respectively. FDP is a state-of-the-art single-core prefetcher aggressiveness control engine.

From Figure 1 it is observed that for the baseline aggressive stream prefetcher, L2 prefetch-fraction varies across applications. The stream prefetcher generates fewer prefetch requests for applications like *astar, bzip2, milc*, and *omnetpp*, than it does for applications with streaming behavior, such as *apsi, libq, leslie3d, lbm, wup, and stream* benchmark. For *astar, bzip2, milc*, and *omnetpp*, L2 prefetch-fraction is less than 10% and their L2 prefetch-accuracy is also low (around 5%). However, with an increase in L2 prefetch-fraction values (along x-axis), L2 prefetch-accuracy also increases. A linear plot across all the data points in the figure shows a positive correlation. In particular, the plot shows 0.76 on the Pearson correlation coefficient [Sharma 2005] and 0.68 on the Spearman rank-correlation coefficient metric [Sharma 2005]. A similar observation

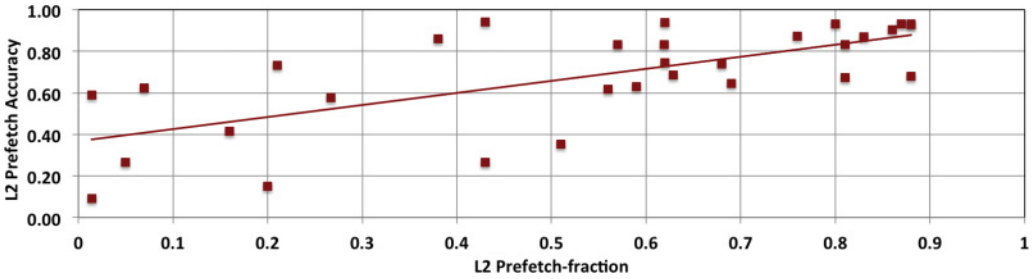

Fig. 3. Scatter plot showing positive correlation between L2 Prefetch-fraction versus L2 Prefetch-accuracy for benchmarks under AMPM prefetching: Pearson correlation coefficient: 0.68 and Spearman rank correlation: 0.65.

can be made from Figures 2 and 3 for FDP and Access Map Pattern Matching (AMPM) prefetchers, respectively [Ishii et al. 2009]. *Altogether, the three plots indicate a strong positive correlation between L2 prefetch-fraction and L2 prefetch-accuracy: the lesser the fraction of prefetch requests generated, the less-likely that they are useful.*[2] *Therefore, we approximate the usefulness of prefetching (prefetch-accuracy) using the L2 prefetch-fraction metric. The significance of our observation of the correlation is that it enables a simple method of measuring usefulness of prefetching. Measuring prefetch-fraction requires only two counters and a simple logic* (Section 4.1).

3.2. Correlation between Prefetcher-caused Delay and Prefetch-fraction

High performance memory controllers like First Ready-First Come First Serve (FR-FCFS) [Rixner et al. 2000] and Prefetch-Aware DRAM Controller (PADC) [Lee et al. 2008] re-order requests to exploit row-buffer locality, and maximize throughput. When a memory controller prioritizes row-hits over row-conflicts, prefetch requests tend to get prioritized over demands. This is because an earlier request opens a row and the subsequent sequence of prefetch requests to the same row exploit row-buffer locality. Therefore, the average service time (LLC miss-penalty or roundtrip latency between LLC and DRAM) of prefetch requests is shorter than that of demands. This disparity in service times between prefetch and demand requests grows proportionally with increase in the ratio of total prefetches to total demand requests at the LLC-DRAM interface.

Figure 4 illustrates this observation for a 16-core workload that consists of applications, such as *vpr, streamcluster, wup, mcf, blackscholes, hmmer, stream, lbm, apsi, sphinx, leslie3d, mesa, vortex, perlbench, astar*, and *wrf*, which have mixed prefetch-friendliness characteristics (refer to Table III). The x-axis represents execution time in intervals of 1 million LLC misses, and the y-axis represents (i) the ratio of total prefetches to that demands (P/D) as well as (ii) the ratio of average miss service times of demands to prefetch requests (AMST (D/P)). From the figure, we observe that with aggressive stream-prefetching that uses no prefetcher throttling, the total prefetches at the LLC-DRAM interface is always higher than that of total demands.

From Figure 4, we observe that the ratio of average miss service times of demands to prefetch requests increases/decreases with the increase/decrease in the ratio of total

[2]If the access pattern is conceivable to the prefetcher, but the application possesses only few prefetch-able contexts, the prefetcher generates fewer but accurate prefetch requests. For a stream prefetcher, this scenario happens when there is a small but regular stream (ex, soplex), while for AMPM, this scenario happens when the smaller number of prefetch-able cache blocks (though irregular) repeat and when AMPM prefetcher is able to capture them (ex, bzip2, and omnetpp). However, as we observe from the plots, the overall correlation is strong.

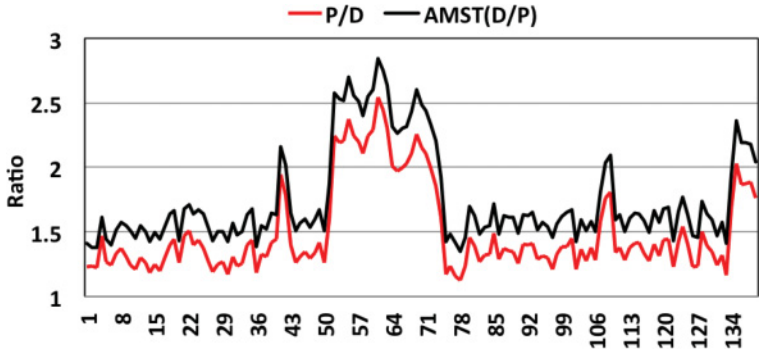

Fig. 4. Correlation between the ratio of LLC miss service times of demand to prefetch requests increases with increase in the ratio of total prefetch requests to that of demands in the system. The x-axis represents the execution time of the workload in intervals of 1 million LLC misses. The y-axis represents (i) the ratio of Average Miss Service Time (AMST) of demands to prefetch requests and (ii) the ratio of total prefetch to demand requests (P/D).

prefetch requests to total demands. In other words, as the ratio of total prefetches to total demands increase, the degree of interference induced on demands (observed in terms of average LLC miss service times of demands) by prefetch requests also increases. Statistically, we observe a very strong positive correlation (0.97 on Pearson coefficient[3]) between ratios of the two quantities. We also observe (i) a strong positive correlation (0.96 on Pearson coefficient) between the ratio of aggregate prefetch to demand requests and the ratio of bandwidth consumed by prefetch to demands and (ii) a strong correlation (0.95 on Pearson coefficient) between the ratio of bandwidth consumed by prefetch to demands and the ratio of average service times of demand to prefetch requests.[4] However, estimation of latency gives a direct indication on prefetcher-caused interference; we use it in our study. From these two observations, we therefore conclude that the interference caused by prefetch requests on demands can be approximated using the ratio of aggregate prefetch to demand requests at the LLC-DRAM interface.

Significance of the Two Correlations: The correlations observed are significant as they enable the introduction of a simple metric, prefetch-fraction, which, as a single metric, when measured at the private L2-LLC and shared LLC-DRAM interface, abstracts both prefetch-usefulness and prefetcher-caused interference. In particular, the correlation in Figure 4 paves a simple way of capturing interference (at the shared LLC-DRAM interface) without having to measure multiple metrics, such as bandwidth needed, consumed, and cycles stalled for each application at the different shared resources, such as DRAM banks, bus, and rows. Measuring prefetch-fraction is easy to implement with a modest hardware cost (Section 6.8).

Altogether, *prefetch-fraction as a metric captures both the usefulness of prefetching (in terms of prefetch-accuracy) to an application when measured at the private L2-LLC interface, as well as the prefetcher-caused interference (in terms of prefetcher-induced delay) when measured at the LLC-DRAM interface*.

[3]We obtain correlation from all the 16-core workloads.

[4]The correlation is strong, because as the total prefetch requests in the system increases, they tend to occupy the shared resources (such as DRAM bus, banks, and rows) more as compared to demand requests, and consequently, the delay incurred in the service of demand requests grows with increase in the bandwidth consumed by prefetch requests.

4. BAND-PASS PREFETCHING

In this section, we present Band-pass prefetching, a dynamic prefetcher aggressiveness control mechanism, to manage prefetcher-caused interference in multi-core systems that exploit the two correlations as discussed in Section 3. Our mechanism is interval based. Our inference drawn in one interval is used to select whether to apply throttling in the subsequent interval.

4.1. High-pass Prefetch Filtering

In Section 3.1, we showed that L2 prefetch-accuracy strongly correlates with L2 prefetch-fraction. To leverage this correlation, we compute prefetch-fraction for an application at runtime. If the measured value of prefetch-fraction is less than a certain threshold, the component probabilistically issues/allows prefetch requests to go to next level. That is, only one in every *16th* prefetch requests are issued to the next level, while the rest of the prefetch requests are dropped.[5] Since this filter component issues all the generated prefetch requests to the next level only when the prefetch-fraction is higher than the threshold, we call this component *High-pass* prefetch filter (analogous to high-pass filter, which allows only signal frequencies that are higher than a threshold to pass through).

Measuring Prefetch-fraction: For measuring prefetch-fraction, we use *two* counters: *L2PrefCounter* and *TotalCounter*. *L2PrefCounter* records the L2 prefetch requests while *TotalCounter* holds the total requests (demands and prefetches from L1 and L2 caches) at the L2-LLC interface. At the end of every interval, the ratio of the two counters gives the L2 prefetch-fraction value, which is stored in a register called *Prefetch-fraction* register. After computation of prefetch-fraction at the end of interval, only the counters are reset; we use the content of *Prefetch-fraction* register to make prefetch issue decisions for the next interval.

4.2. Low-pass Prefetch Filtering

In a multi-core system, memory requests of one or more applications interfere with the others at the shared last level cache and off-chip memory access. We have observed that the LLC miss service time of demand requests (and therefore the likely stall-time of the missing processor) increases with the number of prefetch requests. Ideally, we expect the average service time of demand requests to be lesser than that of prefetch requests as demands are likely to stall the processor when compared to prefetch requests. Therefore, we propose a filter at the shared LLC-DRAM interface that controls prefetcher aggressiveness when the average miss service time of demands *exceeds* that of prefetch requests.

Testing this condition alone, however, is not sufficient because the ratio of prefetch to demand requests and their relative bandwidth consumption are also strongly correlated (recall from Section 3 a correlation of 0.96 on Pearson coefficient). Therefore, controlling the prefetcher aggressiveness only by comparing the ratio of average miss service times of demands and prefetch requests alone can lead to conservatively controlling the prefetchers while the prefetch requests do not consume much bandwidth (and do not cause interference). Therefore, our mechanism also checks if the total prefetch requests *exceed* the demands when the average service time of demand requests exceeds that of prefetch requests. Altogether, the condition to apply prefetcher

[5]By dropping a prefetch request, we refer to not issuing it to the next level (from L2 to LLC). We drop prefetch requests instead of adjusting the prefetcher-configuration in distance and degree. We observe dropping prefetch requests performs better than the latter because, dropping reduces prefetch issue-rate quicker and also issues fewer prefetch requests.

Table I. Set of Counters Used in Estimation of Average Miss Service Time

Counter name	Purpose	Counter name	Purpose
FirstAccess	Time of the first miss in that interval	ElapsedCycles (intermediate)	Cycles spent on servicing *TotalMisses*
LastAccess	Time of last completed miss	TotalElapsedCycles (at end of interval)	Total cycles spent on servicing *TotalMisses*
OutstandingMisses	Current in-flight misses	AvgServiceTime	Holds the avg. service time
TotalMisses	Total completed misses in that interval		

aggressiveness control is given by Equation (1), where AMST (D or P) refers to Average Miss Service Time of demand or prefetch requests. TP and TD refer to the total prefetches and demands at the LLC-DRAM interface, respectively.

$$\text{if}\left(\left(\frac{\text{AMST}(D)}{\text{AMST}(P)} > 1\right)\text{AND}\left(\frac{TP}{TD} > 1\right)\right). \tag{1}$$

In the condition mentioned in Equation (1), $\frac{TP}{TD}$ is a function of their relative bandwidth consumption, and their relationship is $\frac{TP}{TD} = F(\frac{BWCP}{BWCD})$, which we found as $\frac{TP}{TD} = \alpha(\frac{BWCP}{BWCD})$, where the exact value of α is around 1. While we explore various values for α, it is hard to fix its exact value. Hence, we approximate it to 1 and check only if $\frac{TP}{TD} > 1$ alone, which is very simple to implement in hardware: a 16-bit comparator. Ideally, the optimal value of this threshold ratio depends on memory scheduling and workload characteristics (total demand and prefetch requests), which in turn affect the delay induced on demand and prefetch requests.[6] In the following subsection, we explain our mechanism that estimates the average miss service times of demands and prefetches followed by describing the process of collecting prefetch-fraction metric for applications at the shared LLC-DRAM interface.

4.3. Estimation of Average Miss Service Time

We propose a mechanism that employs a set of counters and comparator logic to estimate the average service times of demand and prefetch requests. Table I lists the set of counters and their purpose. Algorithm 1 describes our mechanism.

Explanation: The algorithm is triggered either on a miss[7] at the LLC or when a miss is serviced back from the DRAM. The use of *FirstAccess, LastAccess, OutstandingMisses*, and *TotalMisses* counters ensure that the overlapping of misses is taken into account while estimating average miss service times. Precisely, the time gap (in cycles) between *LastAccess* and *FirstAccess* counters when *OutstandingMisses* counter is zero indicates the cycles that have elapsed while servicing *TotalMisses* number of misses.

At the end of an interval, average service time is estimated. Computing the total cycles elapsed during that interval depends on the value of *OutstandingMisses* counter, which indicates the number of outstanding misses that started in that interval but, have not yet finished. If the value is not *zero*, our algorithm makes an approximation. It sets *LastAccess* counter value to the clock cycle at which the interval ends. Then, it adds *OutstandingMisses* counter value to *TotalMisses*. The difference between *LastAccess*

[6]When the overall number of prefetches and demands are lower than what the system can actually handle, the ideal solution would just be to prioritize demands ahead of prefetches, instead of throttling prefetching. Implementing this solution requires identifying when bandwidth becomes excessively available/saturated and re-ordering requests accordingly at the memory controller (beyond the scope of this work). However, we did not observe this kind of scenario in our experiments.

[7]In this subsection, by miss we refer to either prefetch or demand miss alone. Since we are only interested in their service times, we ignore writebacks. Note that we use separate circuits of the same algorithm for prefetch and demand requests.

ALGORITHM 1: Estimation of Average Miss Service Time

1: **On a new Miss at LLC**
2: *OutstandingMisses*++
3: *FirstAccess*=CurrentClock if it is Reset
4: **When a Miss is Serviced back from DRAM**
5: −−*OutstandingMisses*
6: *LastAccess*=CurrentClock
7: *TotalMisses*++
8: **if** *OutstandingMisses == 0* **then**
9: *ElapsedCycles*=(*LastAccess-FirstAccess*)
10: *TotalElapsedCycles*+=*ElapsedCycles*
11: *FirstAccess*=*LastAccess*=0 //Reset
end
12: **At the end of an Interval**
13: **if** *OutstandingMisses*≠ 0 **then**
14: *TotalMisses*+=*OutstandingMisses*
15: *LastAccess*= CurrentClock
16: *ElapsedCycles*=(*LastAccess-FirstAccess*)
17: *TotalElapsedCycles*+=*ElapsedCycles*
18: *FirstAccess*=Beginning of Next Interval
end
else
19: *FirstAccess=zero* //Reset
end
20: AverageServiceTime=$\frac{\text{TotalElapsedCycles}}{\text{TotalMisses}}$
21: *TotalMisses=ElapsedCycles=TotalElapsedCycles*=0

and *FirstAccess* is added to *TotalElapsedCycles* counter. Finally, *FirstAccess* counter is set to the beginning of the next interval so that the residual cycles of the outstanding misses are accounted for in the subsequent interval. On the other hand, if the value of *OutstandingMisses* is *zero*, the elapsed cycles already computed (line numbers 8–11 in the algorithm) gives the total elapsed cycles while servicing *TotalMisses* number of misses in that interval.

In the algorithm, steps between lines 13 and 18 ensure that the cycles spent by the outstanding misses are accounted for in two successive intervals. That is, first, in the interval in which the misses start (and remain outstanding) and second (the residual cycles), in the subsequent interval in which they finish. Though we have not fully included the cycles spent by those outstanding misses in either the current or the next interval, the error due to this approximation is marginal since the length of the interval is large: millions of clock cycles required to cover an interval size of 1 million LLC misses.

4.4. Selecting the Application to Perform Prefetcher Aggressiveness Control

When band-pass prefetching detects interference on demands by prefetches (using Condition 1), it decides to control the prefetcher of the application that *issues the highest global fraction* of L2 prefetch requests. This decision is inline with our observation presented in Section 3.2: prefetcher-caused interference (delay on demands) increases proportionally to the ratio of total prefetch to demand requests. Hence, the application with the highest L2 prefetch-fraction causes the *most interference*. Therefore, Low-pass component issues only *50%* of the prefetch requests of this application. That is, only one in every *second* prefetch request is issued to the next level. Prefetchers of other applications are allowed to operate in aggressive mode. Similarly, when Band-pass

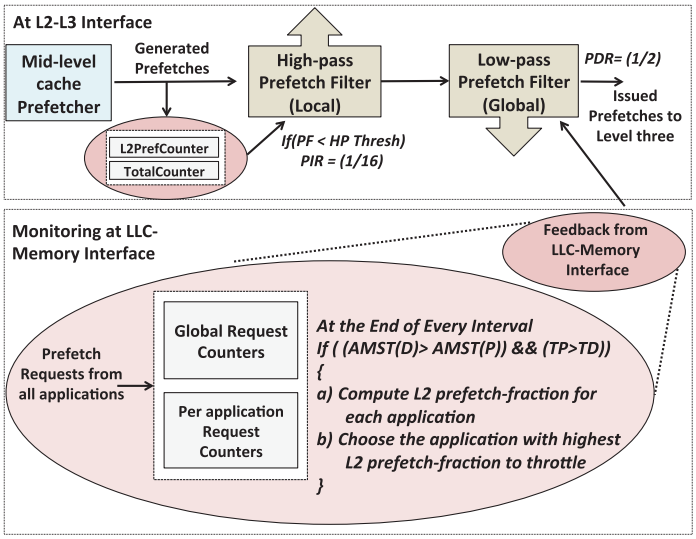

Fig. 5. Schematic diagram of Band-pass Prefetching. PI(D)R: Prefetch Issue (Drop) Rate, PF: prefetch-fraction and HP Thresh: High-pass Threshold.

prefetching detects prefetcher-caused interference to be low, it allows all prefetchers to operate in aggressive mode.

Reason for controlling the prefetcher of only one application: When our mechanism infers interference, the goal is to decrease the overall number of prefetch requests in the system. In order to achieve this, we drop the prefetch requests of the application that issues the highest fraction of prefetch requests, because on some workloads we observe only one application to have a dominating prefetch-fraction value. Therefore, in this case, controlling the most interfering prefetcher is the desired decision (recall that there is a strong correlation between prefetch-fraction and prefetcher-caused interference). On other workloads, more than one application dominates the global prefetch-fraction values. In this case, selecting only one application is still desirable, since our mechanism would control the prefetcher of more than one application across intervals (that is, dropping prefetch requests of one application, say app. A, makes the global prefetch-fraction of another application, say app. B, dominate in the subsequent interval. Consequently, Band-pass prefetching would choose app. B to control in the interval after that). Hence, we select only *one* application (prefetcher) to control when interference is detected.

Measuring Global Prefetch-fraction of Applications: The method of measuring global prefetch-fraction at the shared LLC-DRAM interface is similar to High-pass prefetching except for the fact that the total requests measured by Low-pass correspond to the requests from all applications at the LLC-DRAM interface.

4.5. Overall Band-pass Prefetcher

Figure 5 shows the logical diagram of our proposed band-pass prefetching mechanism. The high-pass and low-pass filters operate independently. During the interval, the high-pass filter computes the local prefetch-fraction of each application at the L2-LLC interface while the low-pass filter computes the global prefetch-fraction of each application at the LLC-DRAM interface. We define this interval in terms of misses at

Table II. Baseline System Configuration

Processor	four-way OoO, 3.3GHz, ROB: 128, RS : 36, LD/ST: 36/24
Branch predictor	TAGE, 16-entry RAS
IL1 and DL1	32KB, LRU, next-line prefetch; ICache: two-way, DCache: eight-way
L2 (unified)	256KB, 16-way, DRRIP [Jaleel et al. 2010], 14-cycle, MSHR: 32-entry
LLC (unified and shared)	16MB, 16-way, PACMAN [Wu et al. 2011], 24-cycle, 256-entry MSHR, 128-entry WB
Interconnect	16 × 4 crossbar, VPC [Nesbit et al. 2007] based arbitration, eight cycles latency
DRAM controller channels-rank-banks Transaction and Channel Queue	Open row, FR-FCFS with prefetch prioritization [Lee et al. 2008] four controllers (one per channel) for 16-core (4-1-8) for 16-core (128,32)
DDR3 parameters	(11-11-11), 1,333 MHz, IO Bus frequency: 1,066MHz

the last level cache. From experiments, we fix 1 million LLC misses as the interval size. Throttling components of the two filters are triggered at the end of the interval. From the feedback collected about the local and global prefetch-fraction during the current interval, if their respective prefetch-fraction is below or higher than their respective thresholds, the two filters control the prefetch issue rate of the prefetchers in the next interval. Note that Band-pass prefetching does not require any modification to cache tag arrays or Miss Status Holding Registers (MSHRs) [Kroft 1981]. Estimation of average service times and computation of prefetch-fraction all lie outside the critical path.

5. EXPERIMENTAL SETUP

5.1. Baseline System

We use the cycle-accurate BADCO [Velasquez et al. 2012] x86 CMP simulator that models a four-way OoO core with a cache hierarchy of three levels. Level 1 and Level 2 caches are private. The last level cache and the memory bandwidth are shared by all cores. Similar to prior studies [Wu et al. 2011; Seshadri et al. 2015; Ebrahimi et al. 2009; Panda and Balachandran 2015; Panda 2016], we model bank-conflicts but with fixed access latency across all banks. Cache line size is 64 bytes throughout the hierarchy and we do not enforce inclusion across cache levels. Our prefetcher model is as described in Section 2.1. We model a 16 × 4 crossbar network. We faithfully model latency and contention in the network. A Virtual Private Cache (VPC) [Nesbit et al. 2007] based scheduler arbitrates requests from L2s to LLC. In our simulated 16-core system, we use *four* independent memory controllers (one per channel), essentially with a configuration of one memory controller for four cores. We use page-interleaved mapping of addresses across channels and to map pages to banks, we use XOR-interleaved mapping. Other system parameters are available in Table II.

5.2. Benchmarks and Workloads

We use SPEC CPU 2000, SPEC CPU 2006 [Henning 2006], and PARSEC [Bienia 2011] benchmark suites totaling 34 (31 from SPEC and 3 from PARSEC) and one stream benchmark. Similar to prior studies [Ebrahimi et al. 2009; Panda and Balachandran 2015; Panda 2016], we classify benchmarks (also referred to as applications in our discussions) based on their IPC improvement over no prefetching when run alone (Table III). We construct four types of workloads, namely, *mixed-type, highly prefetch-friendly, medium prefetch-friendly*, and *prefetch-unfriendly*. Table IV lists each workload type and its construction methodology using the benchmarks as classified

Table III. Classification of Benchmarks

Category	Benchmarks
Highly prefetch-friendly (class A) IPC [≥10%]	apsi, cactusADM, leslie3d, libquantum (libq), lbm, sphinx (sph), STREAM
Medium prefetch-friendly (class B) IPC [≥2%,<10%]	blackscholes, facesim, hmmer, mcf, mesa, vpr, wupwise (wup), streamcluster (strclust)
Prefetch-unfriendly (class C) IPC [±2%]	art, astar, bzip2, deal, gap, gobmk, gcc, gzip, h264ref, milc, omnetpp, perlbench, soplex, twolf, vortex, wrf

Table IV. Workload Types and Their Composition

Type	#Benchmarks from class (A,B,C)	#Workloads
Mixed prefetch-friendly	(5,5,6), (5,6,5) , (6,5,5)	12 each
(Type A) Highly prefetch-friendly	(10,3,3)	20
(Type B) Medium prefetch-friendly	(3,10,3)	20
(Type C) Prefetch-unfriendly	(3,3,10)	20

in Table III. In total, we study 96 16-core multi-programmed workloads. In our experiments, we use the portion of benchmarks between 12 and 12.5 billion instructions. In that phase, the first 200 million instructions of each benchmark warm up all the hardware structures. The next 300 million instructions are simulated. Simulations are run until all benchmarks finish 300 million instructions. If a benchmark finishes execution, it is re-wound and re-executed. Statistics are collected only for the first 300 million instructions.

6. RESULTS AND ANALYSIS

We first present the performance results of High-pass prefetching, our local component (at the private L2 to LLC interfaces) that throttles prefetch requests of an application based on its prefetch-fraction. Then, we present the performance results of Band-pass prefetching that consists of both High-pass and Low-pass components. Throughout our study, we use Harmonic mean of Speedup (HS) [Luo et al. 2001] since it balances both system fairness and throughput.

6.1. Performance of High-pass Prefetching

High-pass prefetching dynamically computes prefetch-fraction of an application at its private L2-LLC interface to infer usefulness (accuracy) of prefetching. If the computed prefetch-fraction value is below a threshold, it begins to control the number prefetch requests issued. The goal of high-pass prefetching is to drop useless prefetch requests and avoid interference caused by them. Here, we use that observation to study the sensitivity of prefetch-fraction threshold values on High-pass prefetching. Figure 6 shows the performance (in HS) of High-pass prefetching across 12 High-pass threshold values (between 9% and 42% in steps of 3%). Performance is normalized to the baseline that implements aggressive stream prefetching without prefetcher throttling. Results are averaged (geometric-mean) across workload types.

Overall analysis: Between thresholds 9% and 21%, there is marginal improvement in performance as compared to aggressive stream prefetching without prefetcher throttling. By cutting down useless prefetch requests, High-pass prefetching attempts to mitigate the interference caused by such useless prefetch requests. While on some individual workloads performance gain is comparable to the baseline, there is marginal improvement on most workloads across workload types. It should be noted that High-pass prefetching achieves marginal improvement while saving the number

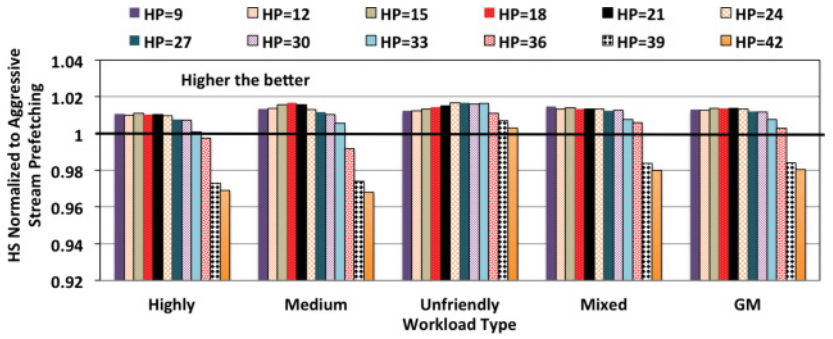

Fig. 6. Performance improvement of High-pass prefetching over Aggressive prefetching across High-pass thresholds. GM: Geometric Mean from all 96 workloads.

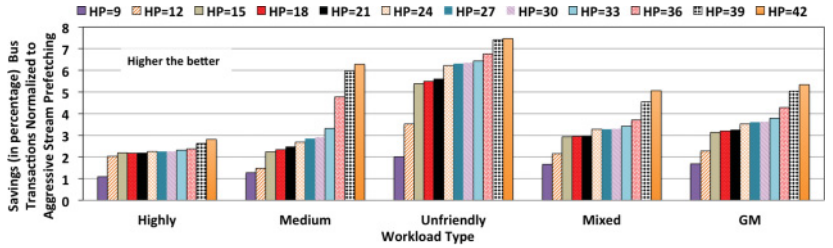

Fig. 7. Savings (in percentage) in bus transactions due to High-pass prefetching (across High-pass thresholds) as compared to aggressive stream prefetching that implements no prefetcher throttling. GM: Geometric Mean from all 96 workloads.

of bus transactions. Figure 7 shows the percentage savings in bus transactions due to High-pass prefetching.

When the threshold is increased beyond 21%, however, performance begins to drop. The impact is higher on highly and medium prefetch-friendly workloads while the impact is less on prefetch-unfriendly workloads. Therefore, as the threshold increases beyond 21%, useful prefetch requests are also dropped down by High-pass (recall from Section 3.1 that prefetch-accuracy increases with increase in prefetch-fraction). Accordingly, we set the High-pass threshold to 21%. Figure 7 shows the percentage of bus transactions saved due to such high-pass prefetching.

Impact of High-pass Prefetching on Individual Applications: Dropping prefetch requests does not affect the performance of individual applications except *mesa* (close to 5% on average). This is because *mesa* exhibits varying prefetch-fraction values across successive intervals, and consequently useful prefetch requests are dropped. However, the impact on the overall workload performance is not significant as performance gain in other applications offsets the performance loss. With regard to applications that do not have a strong correlation relationship between prefetch-fraction and prefetch-accuracy, such as *soplex* and *cactusADM* with stream prefetching, and *bzip2, omnetpp, h264ref*, and *soplex* with AMPM prefetching, we do not observe slow-down on these applications. These applications possess small prefetch-fraction values, and therefore the impact on performance is very marginal (or, virtually no impact). Interestingly, *cactusADM* is highly sensitive to prefetching (when run alone), however, in multi-core environments dropping down its useful prefetch request does not show an impact on its performance.

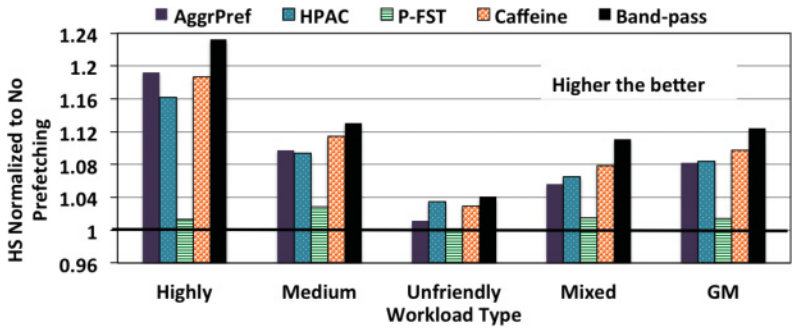

Fig. 8. Performance of prefetcher aggressiveness control mechanisms across workload types. GM: Geometric Mean across all 96 workloads.

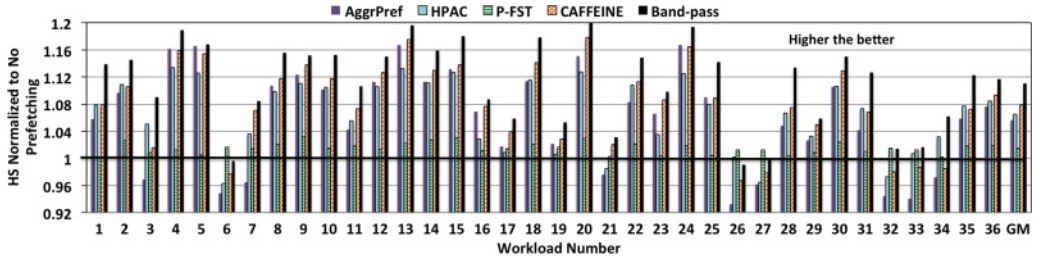

Fig. 9. Performance of prefetcher aggressiveness control mechanisms on Mixed category workload type. GM: Geometric Mean of the 36 mixed-category workloads.

6.2. Performance of Band-pass Prefetching

Figure 8 shows the performance of Band-pass prefetching (referred to as Band-pass in the figures) state-of-the-art prefetcher aggressiveness control mechanisms across different workload types. The last series of bars show the overall (geometric mean) across all 96 workloads we studied. It also shows the performance of aggressive prefetching that does not use prefetcher throttling, state-of-the-art HPAC,[8] P-FST [Ebrahimi et al. 2011], and CAFFEINE. The x-axis represents workload numbers, and the y-axis shows harmonic speedup normalized to *no prefetching*. Over no prefetching baseline, Band-pass achieves 12.4% on average (and up to 21% on a highly prefetch friendly workload), while HPAC, P-FST, and CAFFEINE achieve 8.4%, 1.4%, and 9.7% improvement, respectively. Aggressive prefetching (referred to as AggrPref in the figures) with no prefetcher throttling achieves 8.23% improvement.

General Analysis: On prefetch-friendly workloads, Band-pass prefetching achieves 23.3% performance improvement over no prefetching baseline. Though aggressive prefetching is beneficial, Band-pass is still able to effectively handle interference and achieves higher performance. On medium prefetch-friendly workloads, Band-pass achieves higher performance than the others. On prefetch-unfriendly workloads, HPAC, CAFFEINE, and Band-pass prefetching achieve comparable performance. Overall, Band-pass prefetching *improves performance across different workload types*, and hence, we infer our mechanism is *robust*.

In order to analyze and understand individual mechanisms, we study the performance of mixed-type workloads. Figure 9 shows the performance of Band-pass

[8]We tune the thresholds of HPAC and P-FST to suit the system configuration that we study.

prefetching across the 36 mixed-type workloads in terms of harmonic speedup. Over no prefetching baseline, Band-pass achieves 11.1% on average, and up to 20.5% on workload 20, while HPAC, P-FST, and CAFFEINE achieve 6.4%, 1.5%, and 7.7% improvement, respectively. Aggressive prefetching with no prefetcher throttling achieves 5.6%. In the following paragraphs, we provide an overview of each of these mechanisms and in Section 6.3 we discuss a case study to understand the mechanisms in detail.

Comparative Analysis: When compared to aggressive prefetching, HPAC degrades performance on workloads that benefit from aggressive prefetching (4, 5, 13, 20, and 34). On workloads that suffer highly from aggressive prefetching (3, 6, and 7), HPAC is not able to completely mitigate prefetcher-caused interference. The use of multiple metrics (driven by their thresholds) does not reflect the actual interference in the system and causes HPAC to make incorrect throttling decisions, and makes it less effective. In contrast, Band-pass prefetching is able to retain the benefits of aggressive prefetching as well as effectively mitigate prefetcher-caused interference achieving 4.6% improvement over HPAC.

P-FST's model of determining the most-interfering application and its use of multiple metrics on top of HPAC together lead to incorrect throttling decisions, and forces prefetchers of several applications to a conservative mode. Therefore, P-FST achieves low performance improvement (close to 1.5%) over no prefetching. For the same reason, on workloads where aggressive prefetching is beneficial, P-FST decreases performance. In contrast, Band-pass prefetching achieves higher performance improvement compared to P-FST. On workloads where aggressive prefetching is harmful, Band-pass prefetching achieves either comparable (at most $\pm 2\%$ on workloads 6, 21, 27, and 32) or higher performance improvement (workloads 3 and 7). Overall, Band-pass prefetching achieves 9.6% over P-FST.

CAFFEINE, on workloads in which aggressive prefetching is beneficial (workloads 4, 5, 13, 20, and 30), achieves performance improvement comparable to Band-pass and aggressive prefetching mechanisms. On certain prefetch-friendly workloads (workloads 15, 24, 29, 31, and 35), Band-pass prefetching is still able to achieve higher performance over CAFFEINE thanks to its effective mechanism of detecting interference. However, on workloads that suffer highly due to prefetcher-caused interference (workloads 3, 26, and 34), Band-pass prefetching outperforms CAFFEINE as CAFFEINE is not able to capture prefetcher-caused interference due to its approximate estimation of miss-penalty. Overall, Band-pass prefetching achieves 3.2% improvement over CAFFEINE.

In Summary: Band-pass prefetching is able to retain the benefit of aggressive prefetching as well as effectively manage prefetcher-caused interference. However, state-of-the-art prefetcher aggressiveness control mechanisms are either conservative in cases where aggressive prefetching is actually beneficial (HPAC and P-FST), or do not completely mitigate prefetcher-caused interference (HPAC and CAFFEINE).

6.3. Understanding Individual Mechanisms

In order to gain insights on the individual mechanisms, we discuss a case study of workload 3, which shows the scenario where state-of-the-art HPAC and CAFFEINE do not completely mitigate prefetcher-caused interference. Figure 10 shows the IPC of individual benchmarks normalized to no prefetching.

HPAC: Under HPAC, *libq* slows-down by 13.8% as compared to aggressive prefetching from 1.38 to 1.19 (Figure 10). In this case, useful and timely prefetch requests of *libq* get delayed by memory requests of other applications. Therefore, its prefetch-accuracy drops to around 35% (which is below HPAC's prefetch-accuracy threshold). Hence,

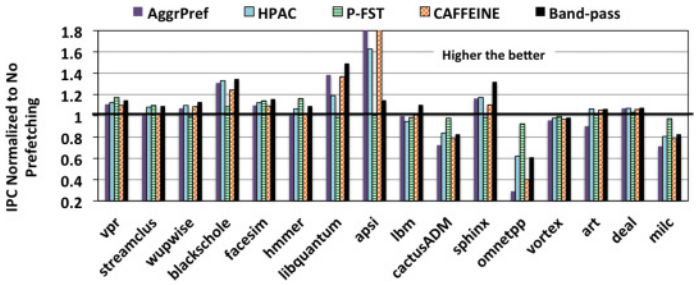

Fig. 10. Normalized IPCs of each application in workload 3.

HPAC throttles-down *libq*'s prefetcher[9] for successive intervals to conservative mode. Under such a scenario where prefetch-accuracy is low, FDP, the local component of HPAC, does not throttle-up the prefetcher as it intends to save bandwidth by throttling-down prefetchers that have low prefetch-accuracy. Therefore, its prefetcher gets stuck in conservative mode, and is not able to exploit the benefit of prefetching. Altogether, HPAC does not detect the interference caused on *libq* and decreases its performance.

P-FST: P-FST's interference models identify applications, such as *cactusADM, libq, apsi, deal*, and *lbm* to be interfering, and conservatively throttle-down their prefetchers on most intervals. Applications, such as *hmmer, facesim*, and *vpr* improve on their performance while most others do not. Since it throttles-down most of its prefetchers, only few benchmarks are able to exploit the benefit of prefetching. Hence, P-FST achieves only marginal increase in performance as compared to no prefetching.

CAFFEINE: CAFFEINE observes positive system-wide net-utility due to prefetching on this workload. This is because CAFFEINE's approximate miss-latency model overestimates the cycles saved due to prefetching. Hence, applications with high prefetch-accuracy, such as *apsi* (96%) and *lbm* (83%), bias system-wide net-utility metric in favor of prefetching. Therefore, though *apsi* consumes high bandwidth, CAFFEINE does not detect interference due to *apsi* and does not throttle-down its prefetcher on most intervals. From Figure 10, we observe that applications such as *omnetpp, milc*, and *cactusADM* suffer slow-down due to interference.

Band-pass Prefetching: Band-pass prefetching computes prefetch-fraction of applications at the shared LLC-DRAM interface to identify the most interfering application. Using prefetch-fraction, it effectively identifies *apsi* as the most-interfering application, and throttles-down its prefetcher. Though the normalized IPC of *apsi* decreases from 2.05 to 1.2, Band-pass prefetching improves the IPCs of applications, such as *libq, omnetpp, sphinx, art, hmmer*, and *lbm*. In doing so, Band-pass prefetching favors both system fairness and throughput. Overall, Band-pass prefetching improves the performance of this workload by 13% as compared to aggressive prefetching, while HPAC and CAFFEINE improve performance by 8% and 5%, respectively.

6.4. Impact on Average Miss Service Time

Band-pass prefetching uses the ratio of average miss service times of demands and prefetches as one of its throttling conditions (Equation (1)). It attempts to decrease the total number of prefetch requests in the system as compared to demands. In doing so, Band-pass prefetching *reduces the interference caused on demands by prefetches*

[9]HPAC also observes high value of BWNO for *libq*. Using the two metrics, HPAC's global component throttles-down its prefetcher (as mentioned in Section 2.2).

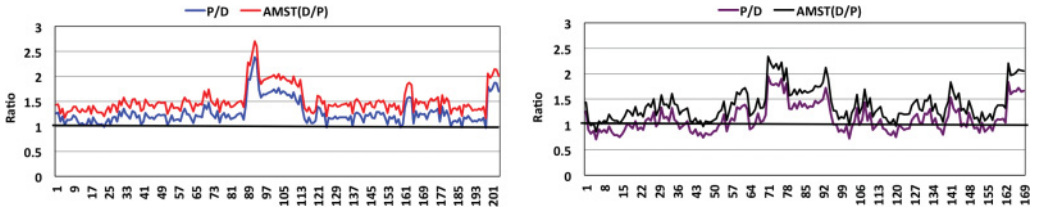

Fig. 11. Correlation between the ratio of prefetch to demand requests (P/D) and the ratio of LLC miss service times of demand to prefetch requests (AMST (D/P)) in the system under (a) Aggressive Prefetching with no prefetcher aggressiveness control (left) and (b) Band-pass prefetching (right). The x-axis represents the execution of the workload in intervals of 1 million LLC Misses. The y-axis represents the ratio of (P/D) and AMST (D/P).

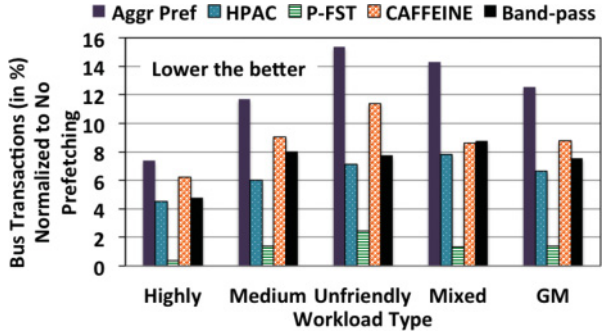

Fig. 12. Increase in Bus Transactions as compared to no prefetching.

in terms of their LLC miss service times, which in turn translates to performance improvement.

Figure 11 shows the ratio of prefetch to demand requests and the ratio of average memory service times of demand to prefetches during the execution of workload *three* with aggressive prefetching and Band-pass prefetching, respectively. The x-axis represents the intervals, which is of 1 million LLC Misses, while the y-axis represents the ratio of the two quantities. As can be seen from Figure 11, the ratio of average memory service time of demand to prefetch requests AMST (D/P) is higher in Aggressive Prefetching with no prefetcher aggressiveness control as compared to Band-pass prefetching. The average AMST (D/P) on this workload with aggressive prefetching is 1.51, which becomes 1.35 under Band-pass prefetching. That is, Band-pass prefetching reduces the average service time of demands by 10.6%. Band-pass prefetching effectively identifies interference happening due to prefetches by checking (P/D) as mentioned in Condition 1. Overall, as compared to aggressive prefetching, Band-pass prefetching reduces the ratio of average service times of total demands to prefetch requests on average from 2.0 to 1.64, while increasing the average service time of prefetch requests by 9.5%.

6.5. Impact on Off-chip Bus Transactions

Figure 12 shows the percentage increase in bus transaction due to prefetching as compared to no prefetching. Aggressive prefetching increases bus transactions by 14.3% while P-FST shows the least increase (only 1.3%) because of its conservative prefetcher throttling as described in Section 6.2. As compared to aggressive prefetching, Band-pass prefetching reduces the bus transactions by 5.55% while achieving better performance of 5.2%. When compared to HPAC and CAFFEINE, Band-pass

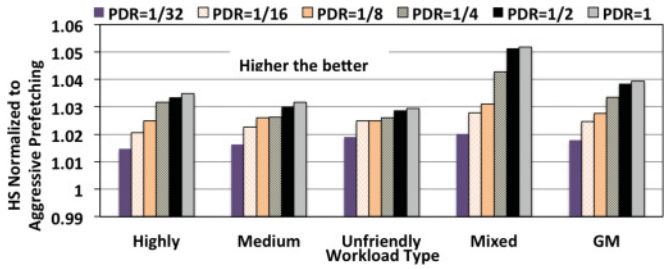

Fig. 13. Sensitivity of Band-pass Prefetching to Prefetch Drop Rate (PDR). GM: Geometric Mean across all 96 workloads.

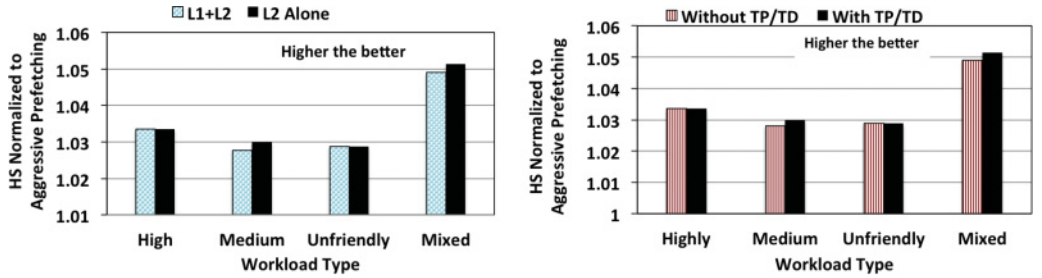

Fig. 14. (a) Impact of including L1 Prefetch Requests (left) and (b) checking TP/TD (right) on Throttling Decisions of condition 1.

achieves performance improvement of 4.6% and 3.2%, respectively, while incurring comparable bus transactions.

6.6. Sensitivity to Design Parameters

Impact of Prefetch Drop Rate: Figure 13 shows the sensitivity of the Low-pass component of our Band-pass prefetching mechanism to prefetch drop rates (represented as PDR) across workload types. In all these experiments, the High-pass prefetch-fraction threshold is fixed at 21% (from Section 4.1). Recall that our mechanism throttles-down prefetch requests by not issuing (dropping) them to the next level. Increase in prefetch drop rate increases the performance up to 5.15% (PDR=1/2), beyond which it saturates. That is, dropping prefetch requests of the most-interfering application beyond 50% does not improve performance further. Therefore, we fix the prefetch drop rate at 1/2 (50%).

Impact of L1 Prefetch Requests on Prefetcher Throttling Decisions: The throttling condition 1 considers only L2 prefetch requests. We study the impact of including L1 Prefetch requests in the throttling decisions. Therefore, TP in TP/TD of condition 1 now represents **(P1+P2)**, where P1 and P2 represents total L1 and L2 prefetch requests, respectively. Figure 14 compares the performance of this design against the former across workload types. Including L1 prefetch requests, *marginally* increases the number of intervals in which TP/TD is greater than *one*, and hence, the number of intervals in which prefetcher aggressiveness control is applied. On workloads where aggressive prefetching is harmful, this design marginally increases the performance. However, on workloads that benefit from aggressive prefetching, performance degrades marginally. Overall, there is a very small performance difference between the two designs. *Therefore, we conclude that L1 prefetch requests do not have a significant impact on our mechanism.*

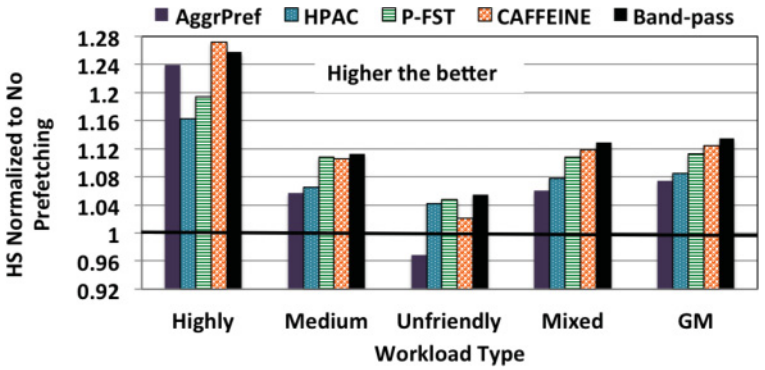

Fig. 15. Performance of prefetcher aggressiveness control mechanisms on systems that implement AMPM prefetching. GM: Geometric Mean.

Impact of TP/TD on Prefetcher Throttling Decisions: Equation (1) presents the conditions under which Band-pass performs prefetcher throttling. To understand the impact of TP/TD on throttling decisions, we ignore the TP/TD comparison in condition 1 and compare only the average miss service times of demands and prefetch requests for making prefetcher control decisions. The right side of Figure 14 shows the performance of this design across workload types. We observe that comparing TP/TD marginally improves the performance on certain workloads, while having no impact on others. Though comparing TP/TD yields a marginal benefit, we observe that without comparing TP/TD, prefetchers in some cases are *conservatively controlled, although their interference due to prefetchers is not significant*. Hence, we include TP/TD in throttling decisions.

6.7. Sensitivity to AMPM Prefetcher

In this section, we evaluate Band-pass prefetching on systems that use AMPM [Ishii et al. 2009] as the baseline prefetching mechanism. We briefly describe AMPM below.

AMPM: AMPM uses a bit-map to encode the list of cache lines accessed in a given region of memory addresses (adapted to 4KB in our study). Each cache line can be in one of the four states: init (initial state), *access* (when accessed by a demand request), *prefetch* (when it is prefetched), or *success* (when the prefetched cache line receives a hit). When there is a demand access to a cache line in a region, AMPM uses the bitmap to extract the stride/offset values from the current demand access. From the prefetchable candidates, if a selected candidate cache line is either in the *access* or *success* state, AMPM issues prefetch requests. In this way, AMPM is able to convert most of the demand requests into prefetch requests.

Figure 15 shows the performance of prefetcher aggressiveness control mechanisms across various workloads in terms of Harmonic Speedup (HS). Band-pass achieves the highest average performance of 13.5% over the no prefetching baseline, while aggressive prefetching with no prefetcher aggressiveness control, HPAC, P-FST, and CAFFEINE achieve 7.4%, 8.47%, 11.2%, and 12.5%, respectively. Interestingly, P-FST achieves higher performance when compared to HPAC. This is because of AMPM's prefetching methodology and P-FST's interference model. P-FST accounts for interference caused by a prefetch or a demand request *only* on the other core's demand requests, and not on the prefetch requests. Therefore, in cases where the demand requests of most applications get converted to prefetch requests (due to AMPM), P-FST does not account interference caused on prefetch requests. Hence, on most intervals, unfairness estimates on individual applications are lower than the unfairness

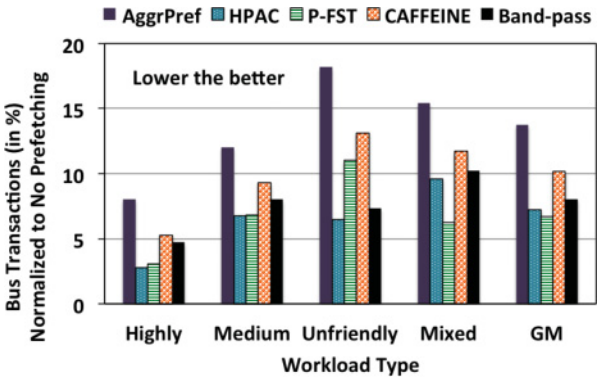

Fig. 16. Increase in Bus Transactions as compared to no prefetching.

threshold, and P-FST allows the prefetchers to run aggressively. On the other hand, HPAC, as before, performs prefetcher throttling based on threshold values of metrics, which is not effective.

CAFFEINE and Band-pass outperform each other on most workloads[10] due to the underlying AMPM prefetching mechanism. AMPM, which issues prefetch requests based on the cache lines status before prefetching them, generates fewer useless cache lines. Consequently, the margin of prefetcher-caused interference is less. CAFFEINE, as mentioned before, observes high positive utility due to prefetching and allows most of the prefetchers to be aggressive. On the other hand, Band-pass prefetching throttles-down more conservatively. (Recall that 50% of prefetch requests are dropped when condition 1 is satisfied.) Hence, there is marginal performance lag as compared to CAFFEINE (around 1.4%). However, on prefetch unfriendly workloads, Band-pass effectively handles prefetcher-caused interference, and achieves 3.3% as compared to CAFFEINE. Overall, Band-pass achieves 1.1% over CAFFEINE.

Figure 16 shows the percentage increase in bus transactions as compared to no prefetching across different workload types. The last series of bars is averaged (geometric avg) across all 96 workloads. Aggressive prefetching increases bus transactions by 13.8%, while P-FST shows the least increase of 6.7%. When compared to aggressive prefetching, HPAC, and CAFFEINE, Band-pass achieves higher performance of 6.1%, 5.05%, 1.1%, while incurring 5.7% (fewer), 0.8% (higher), and 2.1% (fewer) bus transactions, respectively.

Sensitivity to Design Parameters: Figures 17 and 18 show the sensitivity of Band-pass prefetching to design parameters. As observed before, L1 prefetch requests do not have a significant impact on our mechanism. We also make a similar observation on including the ratio of total prefetch to total demand requests (TP/TD) as specified in Equation (1). Figure 18 shows the impact of PDR. Performance increases with an increase in PDR for unfriendly and mixed category workloads. However, for highly prefetch-friendly workloads, increasing PDR marginally decreases performance (up to 1.4%, which is the performance lag of Band-pass as compared to CAFFEINE), since AMPM converts only demands to prefetch requests, and issues fewer useless prefetch requests. Therefore, higher PDRs lead to conservative throttling (when not required). However, higher PDRs (PDR=1/2,1/4, or 1) show a marginal difference in overall performance. From Figures 17 and 18, we observe that the choice of design parameters holds true for AMPM prefetching as well. *Therefore, we conclude that a Band-pass*

[10]Individual workloads not shown due to space limitation.

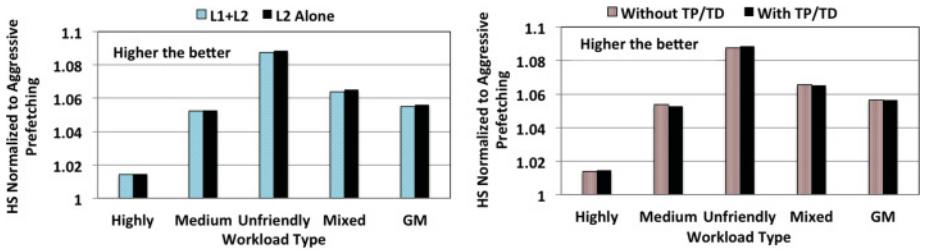

Fig. 17. Impact of (a) Including L1 Prefetch Requests in throttling decisions and (b) Including TP/TD on throttling decisions on systems that implement AMPM prefetching. Performance is Normalized to Aggressive Prefetching. GM: Geometric Mean across all 96 workloads.

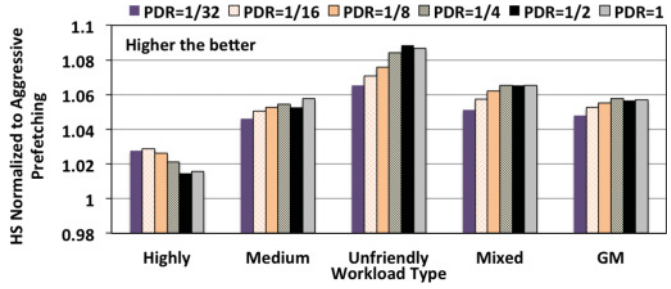

Fig. 18. Sensitivity of Band-pass Prefetching to PDR on systems that implement AMPM prefetching. GM: Geometric Mean across all 96 workloads.

Table V. Hardware Overhead of Band-pass Prefetching

Counter	Purpose	Size
L2PrefCounter	Records L2 prefetches	16-bit
Pref-fraction	Stores prefetch-fraction	16-bit
Drop bit	To drop or Not to drop	1-bit
InsertCounter	For probabilisitic Insertions	4-bit

Counter	Purpose	Size
TotalCounter (High-pass)	Records total requests of an appplication	16-bit
TotalCounter (Low-pass)	Records total requests at shared LLC-DRAM Interface	32-bit

prefetching mechanism does not require parameters tuning across stream and AMPM prefetching mechanisms.

6.8. Hardware Overhead

High-pass and Low-pass prefetching require 53 bits and 37 bits per application, respectively. The first part of Table V shows the counters that are common to both components. High-pass prefetching requires *TotalCounter* per application. However, Low-pass prefetching requires only one *TotalCounter* (32-bit in size) since it measures global prefetch-fraction of applications with only one *TotalCounter*. Hence, we save 16-bit per application for the Low-pass component. To measure interval size in terms of LLC misses, we use a 20-bit counter. Estimation of average miss service times of prefetch and demand requests requires *seven* counters each (Table I). Each counter is 32-bit in size and the total cost amounts to 56 bytes of storage. For a 16-core system, hardware overhead is only 239 bytes, while HPAC, P-FST, and CAFFEINE require about 208KB, 228.5KB, and 204KB, respectively. *Note that Band-pass prefetching does not require any modification to cache tag arrays or MSHR structures*.

6.9. Using Prefetch-accuracy to Control Aggressiveness

The Band-pass prefetching mechanism that we described so far uses prefetch-fraction metric for performing prefetch control decisions. Alternatively, we performed experiments where prefetch control is applied on the application with the least prefetch-accuracy. However, we observed no improvement over aggressive prefetching, since low-accuracy does not correlate/imply high interference. We further experimented with a combination of prefetch-accuracy and prefetch-fraction to throttle down the prefetcher that issues the highest fraction of in-accurate prefetch requests (measured as (1-accuracy) multiplied by prefetch-fraction). Still, we did not observe performance improvement.

6.10. Overall Inference

Overall, Band-pass prefetching achieves higher performance improvement[11] as compared to the other mechanisms across stream and state-of-the-art AMPM prefetchers. Also, the hardware cost of Band-pass is modest, only 239 bytes for a 16-core system without incurring any changes to the existing cache and MSHR designs. Hence, we infer our proposed Band-pass prefetching to be an effective and robust mechanism for managing prefetching in multi-core systems.

6.11. Applying Band-pass Prefetching to Parallel Workloads

To apply Band-pass prefetching to parallel workloads, it is possible to treat each thread or task as an independent application. As with multi-programmed workloads, Band-pass controls the most interfering thread of a parallel application.[12] In addition, Band-pass prefetching could append thread-criticality information [Du Bois et al. 2013; Panda and Balachandran 2013] for performing prefetcher aggressiveness control. However, when the threads of a parallel application share data and work co-operatively (observed by Natarajan and Chaudhuri [2013] and Panda and Balachandran [2012]), it is possible to append Band-pass prefetching with data sharing characteristics for further improving the execution time of parallel applications.

7. RELATED WORK

Prefetching has been extensively studied in the past. Prior works on prefetching focused both on exploiting simple sequential access patterns (e.g., Smith [1978], Jouppi [1990], and Palacharla and Kessler [1994]) in applications as well as on complex, non-sequential access patterns [Joseph and Grunwald 1997; Lai et al. 2001; Nesbit and Smith 2004; Pugsley et al. 2014; Shevgoor et al. 2015; Michaud 2016]. While several works focused on maximizing the benefit of prefetching, some other works studied the interference caused by prefetching both in single-core [Zhuang and Lee 2003; Srinath et al. 2007; Hur and Lin 2006, 2009; Wu et al. 2011; Seshadri et al. 2015] and multi-core [Ebrahimi et al. 2009, 2011; Panda and Balachandran 2015; Jimenez et al. 2015; Panda 2016; Ishii et al. 2012] contexts. In this section, we discuss works that are close to our work.

7.1. Prefetch-filter based Techniques

Several studies have proposed efficient prefetch-filtering mechanisms that attempt to control the number of inaccurate (unused) prefetch requests generated under stream based prefetchers. Zhuang and Lee [2003] propose a filtering mechanism that uses a

[11]With regard to CAFFEINE, though performance gain is marginal, it does so with fewer bus transactions (Figures 12 and 16) and modest hardware cost (Section 6.8) as compared to CAFFEINE.

[12]When all the threads of a parallel application perform proportional work, we may select the prefetcher of one of the threads randomly for prefetcher control.

history table (2-bit counter indexed by Program Counter or cache block address) to decide the effectiveness of prefetching. Prefetch requests are issued depending on the outcome of the counter. Hur and Lin propose Adaptive Stream Detection mechanisms [Hur and Lin 2006, 2009] for effectively detecting short streams using dynamic histograms. All these mechanisms have been developed to improve the efficiency of a prefetcher (in a single-core context). In multi-core systems, prefetching typically causes interference, which these mechanisms cannot capture; they must be augmented with techniques that capture interference. Conversely, our mechanism uses prefetch-fraction to determine both the accuracy (usefulness) of prefetching as well as prefetcher-caused interference.

7.2. Adaptive Prefetching Techniques

Jimenez et al. independently propose [Jiménez et al. 2012; Jimenez et al. 2015] software based approaches to perform dynamic prefetcher aggressiveness control. In their work, Jiménez et al. [2012] attempt to select the best prefetch setting for an application by exploring prefetch utility across different configurations. A simple prefetch utility model reduces the impact of exploration phase on performance. Similarly, Panda proposes Synergistic Prefetcher Aggressiveness Control [Panda 2016], a mechanism that attempts to minimize search space of prefetch configurations across applications that together satisfy system-wide harmonic-speedup goal. On the other hand, our mechanism does not require exploring across prefetch settings to control the prefetchers. Instead, our mechanism controls the number of prefetch requests by selectively dropping them (using prefetch-fraction metric, which infers both prefetch usefulness and interference). In another work, Jimenez et al. [2015] propose a prefetch efficiency metric that measures the benefit of prefetching in proportion to its bandwidth consumption. When the memory bandwidth saturates, the prefetcher with the least prefetch efficiency is turned off. The principal difference from our work is that our model to controlling prefetching is based on the direct impact of prefetching on the service time of the demand requests since our mechanism is hardware based, which also allows fine-grained control of prefetching. Adaptive prefetch control for Banked Shared LLC (ABS) [Albericio et al. 2012] is a prefetcher aggressiveness control mechanism that is proposed for systems where prefetching is employed at the banks of the shared last level cache. Our mechanism can be applied on top of such systems: prefetcher of the individual banks can be treated as a prefetcher-resource and then monitor the requests from each bank. When there is interference, the prefetcher of the bank that issues the highest prefetch-fraction can be throttled-down.

8. CONCLUSIONS

In this article, we propose Band-pass prefetching, a simple and effective mechanism to manage prefetching in multi-core systems. Our solution builds on the observations that a strong correlation exists between (i) prefetch-fraction and prefetch-accuracy and (ii) the ratio of the average miss service times of demand to prefetch requests, and the ratio of prefetch to demand requests in the system. The first observation infers the usefulness (in terms of prefetch-accuracy) of prefetching while the second observation infers the prefetcher-caused interference on demand requests.

Our mechanism consists of two prefetch filter components: High-pass, which is present at the private L2-L3, and a Low-pass component, present at the shared LLC-DRAM interface. The two components independently compute prefetch-fraction of applications at the private L2-LLC and shared LLC-DRAM interfaces. Together, the two components control the flow of prefetch requests between a range of prefetch-to-demand ratios. Experimental results show that Band-pass prefetching achieves 12.4% improvement over the baseline that implements no prefetching. As compared to state-of-the-art

prefetcher aggressiveness control mechanisms, namely, HPAC, P-FST, and CAFFEINE, Band-pass prefetching achieves 4.6%, 9.6%, and 3.2% higher performance, respectively. Further experiments using AMPM prefetcher observe Band-pass prefetching showing similar performance trends: 13.5% improvement over the baseline that implements no prefetching. As compared to state-of-the-art prefetcher aggressiveness mechanisms HPAC, P-FST, and CAFFEINE, Band-pass prefetching achieves 6.1%, 5%, and 1.1%, respectively. Experimental studies demonstrate that Band-pass is effective in mitigating interference caused by prefetchers, and is robust across workload types. All in all, band-pass prefetching achieves higher performance while requiring only a modest hardware cost of less than 240 bytes.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers and the ALF/PACAP team for its valuable feedback on this work.

REFERENCES

Jorge Albericio, Rubén Gran, Pablo Ibáñez, Víctor Viñals, and Jose María Llabería. 2012. ABS: A low-cost adaptive controller for prefetching in a banked shared last-level cache. *ACM Trans. Archit. Code Optim.* 8, 4, (Jan. 2012), Article 19, 20 pages. DOI: http://dx.doi.org/10.1145/2086696.2086698

Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. 2008. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, Washington, DC, USA, 318–329. DOI: http://dx.doi.org/10.1109/MICRO.2008.4771801

Kristof Du Bois, Stijn Eyerman, Jennifer B. Sartor, and Lieven Eeckhout. 2013. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, 511–522. DOI: http://dx.doi.org/10.1145/2485922.2485966

Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2011. Prefetch-aware shared resource management for multi-core systems. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. ACM, New York, 141–152. DOI: http://dx.doi.org/10.1145/2000064.2000081

Eiman Ebrahimi, Onur Mutlu, Chang. J. Lee, and Yale N. Patt. 2009. Coordinated control of multiple prefetchers in multi-core systems. In *Proceedings of the 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. 316–326.

John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. DOI: http://dx.doi.org/10.1145/1186736.1186737

Ibrahim Hur and Calvin Lin. 2006. Memory prefetching using adaptive stream detection. In *Proceedings of the 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 397–408. DOI: http://dx.doi.org/10.1109/MICRO.2006.32

Ibrahim Hur and Calvin Lin. 2009. Feedback mechanisms for improving probabilistic memory prefetching. In *Proceedings of the 2009 IEEE 15th International Symposium on High Performance Computer Architecture*. 443–454. DOI: http://dx.doi.org/10.1109/HPCA.2009.4798282

Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2009. Access map pattern matching for data cache prefetch. In *Proceedings of the 23rd International Conference on Supercomputing (ICS'09)*. ACM, New York, 499–500. DOI: http://dx.doi.org/10.1145/1542275.1542349

Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2012. Unified memory optimizing architecture: Memory subsystem control with a unified predictor. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'12)*. ACM, New York, 267–278. DOI: http://dx.doi.org/10.1145/2304576.2304614

Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, 60–71. DOI: http://dx.doi.org/10.1145/1815961.1815971

V. Jimenez, A. Buyuktosunoglu, P. Bose, F. P. O'Connell, F. Cazorla, and M. Valero. 2015. Increasing multicore system efficiency through intelligent bandwidth shifting. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. 39–50. DOI: http://dx.doi.org/10.1109/HPCA.2015.7056020

Victor Jiménez, Roberto Gioiosa, Francisco J. Cazorla, Alper Buyuktosunoglu, Pradip Bose, and Francis P. O'Connell. 2012. Making data prefetch smarter: Adaptive prefetching on POWER7. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. ACM, New York, 137–146. DOI:http://dx.doi.org/10.1145/2370816.2370837

Doug Joseph and Dirk Grunwald. 1997. Prefetching using Markov predictors. *SIGARCH Comput. Archit. News* 25, 2 (May 1997), 252–263. DOI:http://dx.doi.org/10.1145/384286.264207

Norman P. Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'90)*. ACM, New York, 364–373. DOI:http://dx.doi.org/10.1145/325164.325162

David Kroft. 1981. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA'81)*. IEEE Computer Society Press, 81–87. http://dl.acm.org/citation.cfm?id=800052.801868

An-Chow Lai, Cem Fide, and Babak Falsafi. 2001. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*. ACM, New York, 144–154. DOI:http://dx.doi.org/10.1145/379240.379259

Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N. Patt. 2008. Prefetch-aware DRAM controllers. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, Washington, DC, 200–209. DOI:http://dx.doi.org/10.1109/MICRO.2008.4771791

Fang Liu and Yan Solihin. 2011. Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'11)*. ACM, New York, 37–48. DOI:http://dx.doi.org/10.1145/1993744.1993749

Kun Luo, J. Gummaraju, and M. Franklin. 2001. Balancing throughput and fairness in SMT processors. In *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'01)*. 164–171. DOI:http://dx.doi.org/10.1109/ISPASS.2001.990695

Pierre Michaud. 2016. Best-offset hardware prefetching. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. 469–480. DOI:http://dx.doi.org/10.1109/HPCA.2016.7446087

R. Natarajan and M. Chaudhuri. 2013. Characterizing multi-threaded applications for designing sharing-aware last-level cache replacement policies. In *Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC'13)*. 1–10. DOI:http://dx.doi.org/10.1109/IISWC.2013.6704665

Kyle J. Nesbit, James Laudon, and James E. Smith. 2007. Virtual private caches. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. ACM, New York, 57–68. DOI:http://dx.doi.org/10.1145/1250662.1250671

Kyle J. Nesbit and James E. Smith. 2004. Data cache prefetching using a global history buffer. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA'04)*. IEEE Computer Society, Washington, DC, 96–105. DOI:http://dx.doi.org/10.1109/HPCA.2004.10030

Alan V. Oppenheim, Alan S. Willsky, and S. Hamid Nawab. 1996. *Signals & Systems*. (2nd ed.). Prentice-Hall, Inc., Upper Saddle River, NJ.

S. Palacharla and R. E. Kessler. 1994. Evaluating stream buffers as a secondary cache replacement. *SIGARCH Comput. Archit. News* 22, 2 (April 1994), 24–33. DOI:http://dx.doi.org/10.1145/192007.192014

Biswabandan Panda. 2016. SPAC: A synergistic prefetcher aggressiveness controller for multi-core systems. *IEEE Trans. Comput.* PP, 99 (2016), 1–1. DOI:http://dx.doi.org/10.1109/TC.2016.2547392

B. Panda and S. Balachandran. 2012. CSHARP: Coherence and sharing aware cache replacement policies for parallel applications. In *Proceedings of the 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*. 252–259. DOI:http://dx.doi.org/10.1109/SBAC-PAD.2012.27

B. Panda and S. Balachandran. 2013. TCPT—Thread criticality-driven prefetcher throttling. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. 399–399. DOI:http://dx.doi.org/10.1109/PACT.2013.6618835

Biswabandan Panda and Shankar Balachandran. 2015. CAFFEINE: A utility-driven prefetcher aggressiveness engine for multicores. *ACM Trans. Archit. Code Optim.* 12, 3 (Aug. 2015), Article 30, 25 pages. DOI:http://dx.doi.org/10.1145/2806891

S. H. Pugsley, Z. Chishti, C. Wilkerson, P. F. Chuang, R. L. Scott, A. Jaleel, S. L. Lu, K. Chow, and R. Balasubramonian. 2014. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. 626–637. DOI:http://dx.doi.org/10.1109/HPCA.2014.6835971

Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. 2000. Memory access scheduling. *SIGARCH Comput. Archit. News* 28, 2 (May 2000), 128–138. DOI:http://dx.doi.org/10.1145/342001.339668

V. Seshadri, S. Yedkar, H. Xin, Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. 2015. Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks. *ACM Trans. Archit. Code Optim.* 11, 4 (Jan. 2015), Article 51, 22 pages. DOI:http://dx.doi.org/10.1145/2677956

A. K. Sharma. 2005. *Text Book of Correlations and Regression*. Discovery Publishing House. https://books.google.fr/books?id=obb7_6k6XDQC.

M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti. 2015. Efficiently prefetching complex address patterns. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, 141–152. DOI:http://dx.doi.org/10.1145/2830772.2830793

B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. 2011. IBM POWER7 multicore server processor. *IBM J. Res. Dev.* 55, 3 (May 2011), 1:1–1:29. DOI:http://dx.doi.org/10.1147/JRD.2011.2127330

A. J. Smith. 1978. Sequential program prefetching in memory hierarchies. *Computer* 11, 12 (Dec. 1978), 7–21. DOI:http://dx.doi.org/10.1109/C-M.1978.218016

S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. 2007. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 63–74. DOI:http://dx.doi.org/10.1109/HPCA.2007.346185

Ricardo A. Velasquez, Pierre Michaud, and André Seznec. 2012. BADCO: Behavioral application-dependent superscalar core model. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS XII)*.

C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, Jr., and J. Emer. 2011. PACMan: Prefetch-aware cache management for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. 442–453. http://doi.acm.org/10.1145/2155620.2155672

X. Zhuang and H. H. S. Lee. 2003. A hardware-based cache pollution filtering mechanism for aggressive prefetches. In *Proceedings of the 2003 International Conference on Parallel Processing*. 286–293. DOI:http://dx.doi.org/10.1109/ICPP.2003.1240591

Received November 2016; revised April 2017; accepted April 2017