Prefetch-Aware DRAM Controllers

Chang Joo Lee *Onur Mutlu‡* *Veynu Narasiman* *Yale N. Patt*

High Performance Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-0240

‡Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213-3890

TR-HPS-2008-002
September 2008

This page is intentionally left blank.

Prefetch-Aware DRAM Controllers*

Chang Joo Lee† Onur Mutlu§ Veynu Narasiman† Yale N. Patt†

†Department of Electrical and Computer Engineering
The University of Texas at Austin
{cjlee, narasima, patt}@ece.utexas.edu

§Department of Electrical and Computer Engineering
Carnegie Mellon University
onur@cmu.edu

Abstract

Existing DRAM controllers employ rigid, non-adaptive scheduling and buffer management policies when servicing prefetch requests. Some controllers treat prefetch requests the same as demand requests, others always prioritize demand requests over prefetch requests. However, none of these rigid policies result in the best performance because they do not take into account the usefulness of prefetch requests. If prefetch requests are useless, treating prefetches and demands equally can lead to significant performance loss and extra bandwidth consumption. In contrast, if prefetch requests are useful, prioritizing demands over prefetches can hurt performance by reducing DRAM throughput and delaying the service of useful requests.

This paper proposes a new low-cost memory controller, called Prefetch-Aware DRAM Controller (PADC), that aims to maximize the benefit of useful prefetches and minimize the harm caused by useless prefetches. To accomplish this, PADC estimates the usefulness of prefetch requests and dynamically adapts its scheduling and buffer management policies based on the estimates. The key idea is to 1) adaptively prioritize between demand and prefetch requests, and 2) drop useless prefetches to free up memory system resources, based on the accuracy of the prefetcher. Our evaluation shows that PADC significantly outperforms previous memory controllers with rigid prefetch handling policies. Across a wide range of multiprogrammed SPEC CPU 2000/2006 workloads, it improves system performance by 8.2% on a 4-core system and by 9.9% on an 8-core system while reducing DRAM bandwidth consumption by 10.7% and 9.4% respectively.

1. Introduction

High performance memory controllers seek to maximize throughput by exploiting *row buffer locality*. A modern SDRAM bank contains a *row buffer* that buffers the data of the last accessed memory row. Therefore, an access to the same row (called *row-hit*) can be serviced significantly faster than an access to a different row (called *row-conflict*) [15]. Due to this non-uniform access latency, state-of-the-art memory access scheduling policies such as [42, 27, 14] prefer row-hits over row-conflicts to improve DRAM throughput, thereby improving system performance. However, the problem of DRAM access scheduling becomes more challenging when we take prefetch requests into consideration.

Today's microprocessors employ hardware prefetchers to hide long DRAM access latencies. If prefetch requests are accurate and fetch data early enough, prefetching can improve performance. However, even if prefetch accuracy is high, the full benefit of prefetching may not be achieved based on how the DRAM controller schedules the requests. For example, a demand request can delay a prefetch request that could have otherwise been serviced very fast in DRAM, e.g., if the prefetch request is a row-hit while the demand request is a row-conflict. If the prefetch is useful, delaying it by servicing the row-conflict demand request first may not result in the best performance.

In addition, prefetching does not always improve and can sometimes degrade performance due to two reasons. First, useless prefetch requests unnecessarily consume valuable off-chip bandwidth and fetch useless data that might displace useful data blocks in processor caches. Second, prefetch requests contend for the same resources (e.g., memory request buffer entries and memory bus bandwidth) as demand (load/store) requests issued by a processing core. As a result, a prefetch request can delay a demand request, which could lead to performance degradation especially if the prefetch is useless. If the interference between prefetch requests and demand requests is not controlled, system performance can degrade because either demand requests or useful prefetch requests can be significantly delayed.

*This work is an extended version of the work presented in the 41st International Symposium on Microarchitecture (MICRO-41) [10], *"Prefetch-Aware DRAM Controllers"*. Sections 1, 2, 4, and 6 are siginificantly extended beyond [10]. Almost all sections are revised to clarify the explanations. This submission contains approximately 50% additional new material. Details of the extensions are described in the accompanying cover letter.

Existing DRAM scheduling policies take two different approaches as to how to treat prefetch requests with respect to demand requests. Some policies [37] regard a prefetch request to have the same priority as a demand request. As noted above, this policy can significantly delay demand requests and cause performance degradation, especially if prefetch requests are not accurate. Other policies [7, 11, 5, 31, 32] always prioritize demand requests over prefetch requests so that data known-to-be-needed by the program instructions can be serviced earlier. One might think that always prioritizing demand requests over prefetch requests in the memory controller provides the best performance by eliminating the interference of prefetch requests with demand requests. However, such a rigid demand-over-prefetch prioritization policy does not consider the non-uniform access latency of the DRAM system (row-hits vs. row-conflicts). A row-hit prefetch request can be serviced much more quickly than a row-conflict demand request. Therefore, servicing the row-hit prefetch request first provides better DRAM throughput and can improve system performance compared to servicing the row-conflict demand request first[1].

Figure 1 provides supporting data to demonstrate this. This figure shows the performance impact of an aggressive stream prefetcher [34, 32] when used with two different memory scheduling policies for 10 SPEC 2000/2006 benchmarks. The vertical axis is retired instructions per cycle (IPC) normalized to the IPC on a processor with no prefetching. One policy, *demand-prefetch-equal* does not differentiate between demand and prefetch requests. This policy is the same as the FR-FCFS (First Ready-First Come First Serve) policy that prioritizes requests as follows: 1) row-hit requests over all others, 2) older requests over younger requests [27]. As a result, DRAM throughput is maximized. The other policy, *demand-first*, prioritizes demand requests over prefetch requests. Prefetch requests to a bank are not scheduled until all the demand requests to the same bank are serviced. Within a set of outstanding demand or prefetch requests, the policy uses the same prioritization rules as the FR-FCFS policy. As a result, this policy does not maximize overall DRAM throughput because it prefers row-conflict demand requests over row-hit prefetch requests[2].

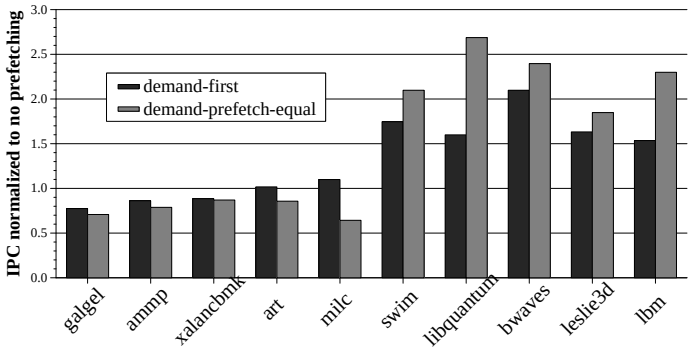


Figure 1. Normalized performance of a stream prefetcher with two different DRAM scheduling policies

The results show that *neither of the two policies provides the best performance for all applications*. For the leftmost five applications, prioritizing demands over prefetches results in better performance than treating prefetches and demands equally. In these applications, a large fraction (70% for demand-prefetch-equal, and 59% for demand-first) of the generated stream prefetch requests are useless. Therefore, it is important to prioritize demand requests over prefetches. In fact, for *art* and *milc*, servicing the demand requests with higher priority is critical to make prefetching effective. Prefetching improves the performance of these two applications by 2% and 10% respectively with the *demand-first* scheduling policy, whereas it reduces performance by 14% and 36% with the *demand-prefetch-equal* policy.

[1] Note that maximizing the number of row-hits provides the highest throughput the DRAM bank can deliver.

[2] For completeness, we also implemented another policy, *prefetch-first*, that always prioritizes prefetch requests over demand requests. This policy provides the worst performance (5.8% IPC degradation compared to demand-first policy) on all the benchmarks.

On the other hand, for the rightmost five applications, we observe the exact opposite behavior. Equally treating demand and prefetch requests provides significantly higher performance than prioritizing demands over prefetches. In particular, for *libquantum*, the *demand-prefetch-equal* policy allows the prefetcher to provide 169% performance improvement, in contrast to the 60% performance improvement it provides with the *demand-first* scheduling policy. This is because prefetch requests in *libquantum* are very accurate (almost 100% of them are useful). Maximizing DRAM throughput by preferring row buffer hits in the DRAM system regardless of whether a memory request is a demand or a prefetch request allows for more efficient bandwidth utilization and improves the timeliness (and the coverage) of prefetches, thereby improving system performance[3]. These results show that DRAM scheduling policies with rigid prioritization rules among prefetch and demand requests cannot provide the best performance and may even cause prefetching to degrade performance.

Note that even though the DRAM scheduling policy has a significant impact on the performance provided by prefetching, prefetching sometimes degrades performance regardless of the DRAM scheduling policy. For example, *galgel*, *ammp*, and *xalancbmk* suffer significant performance loss with prefetching because a large fraction (69%, 94%, and 91%) of the prefetches are not needed by the program. The negative performance impact of these useless prefetch requests cannot be mitigated solely by a *demand-first* scheduling policy because useless prefetches 1) occupy memory request buffer entries in the memory controller until they are serviced, 2) occupy DRAM bandwidth while they are being serviced, and 3) cause cache pollution by evicting possibly useful data from the processor caches after they are serviced. As a result, useless prefetches could delay the servicing of demand requests and could result in additional demand requests. In essence, *useless prefetch requests can deny service to demand requests because the DRAM controller is not aware of the usefulness of prefetch requests in its memory request buffer*. To prevent this, the memory controller should intelligently manage the memory request buffer between prefetch and demand requests.

Our goal in this paper is to design an adaptive DRAM controller that is aware of prefetching. We propose a memory controller that adaptively controls the interference between prefetch and demand requests to improve system performance. Our controller aims to maximize the benefits of useful prefetches and minimize the harm of useless prefetches. To do so, it employs two techniques to manage both memory bandwidth and memory request buffers: based on the runtime behavior (accuracy and timeliness) of the prefetcher, it 1) adaptively decides whether or not to prioritize demand requests over prefetch requests, and 2) decides whether or not to drop likely-useless prefetches from the memory request buffer.

We evaluate our *Prefetch-Aware DRAM Controller* on a wide variety of benchmarks and systems and find that it consistently outperforms previous DRAM controllers that rigidly handle prefetches on both single-core and multi-core (2, 4, and 8-core) systems. Our controller improves the performance of the 55 SPEC 2000/2006 benchmarks by up to 68% (on average 4.3%) compared to the best previous controller on a single core processor. Our mechanism also improves system performance (i.e., weighted speedup) for 54 SPEC workloads by 8.4% on a 2-core system, for 32 workloads by 8.2% on a 4-core system, and for 21 SPEC workloads by 9.9% on an 8-core system while also reducing memory bandwidth consumption by 10.0%, 10.7% and 9.4% for 2, 4, and 8 core systems respectively. We show that our controller is simple to implement and low-cost, requiring only 4.25KB of storage in a 4-core system.

Contributions: To our knowledge, this is the first paper that comprehensively and adaptively incorporates prefetch-awareness into the memory controller's scheduling and request buffer management policies. We make the following contributions:

1. We show that the performance of a prefetcher significantly depends on how prefetch requests are handled by the memory controller with respect to demand requests. Rigid memory scheduling policies that treat prefetches and demands equally or that

[3]Improving DRAM throughput improves prefetch coverage by reducing the probability that a useful prefetch is not issued into the memory system because the memory request buffer is full. We will explain this in more detail in Section 6.1.

always prioritize demands can either cause significant losses in performance or not achieve the best performance for all applications.

2. We propose a low-cost memory controller design that dynamically adapts its prioritization policy between demand and prefetch requests based on how accurate and timely the prefetcher is in a given program phase. This mechanism achieves high performance by improving both DRAM throughput and prefetch timeliness/coverage.

3. We propose a simple mechanism that reduces the interference of useless prefetches with demand requests by proactively removing the likely-useless prefetches from the memory request buffer. This mechanism efficiently reduces the buffer, bandwidth, and cache space resources consumed by useless prefetches, thereby improving both performance and bandwidth-efficiency.

4. We show that the proposed adaptive scheduling and buffer-management mechanisms interact positively. Our *Prefetch-Aware DRAM Controller* (PADC) that comprises both mechanisms significantly improves performance and bandwidth-efficiency on both single-core and multi-core systems. In addition, our proposal is very effective for a variety of prefetching algorithms, including stream, PC-based stride, CZone/ Delta Correlation (C/DC), and Markov prefetching.

5. We comprehensively evaluate the performance and DRAM bus traffic of PADC on various DRAM and last-level cache configurations. We also compare and incorporate PADC with other mechanisms such as hardware prefetch filtering, feedback directed prefetching, memory address remapping, and runahead execution. Our results show that PADC not only outperforms but also complements all of these previous performance enhancement mechanisms.

2. Background

2.1. DRAM Systems and Scheduling

An SDRAM system consists of multiple banks that can be accessed independently. Each DRAM bank comprises rows and columns of DRAM cells. A row contains a fixed-size block of data (usually several Kbytes). Each bank has a *row buffer* (or *sense amplifier*), which caches the most recently accessed row in the DRAM bank. Every DRAM access can only be done by reading (writing) data from (to) the row buffer using a column address.

There are three possible sequential commands that need to be issued to a DRAM bank in order to access data. A memory controller may issue 1) a *precharge* command to precharge the row bitlines, 2) an *activate* command to *open* a row into the row buffer with the row address, and then 3) a *read/write* command to access the row buffer with the column address. After the completion of an access, the DRAM controller can either keep the row open in the row buffer (*open-row* policy) or close the row buffer with a precharge command (*closed-row* policy). The latency of a memory access to a bank varies depending on the state of the row buffer and the address of the request as follows:

1. *Row-hit*: The row address of the memory access is the same as the address of the opened row. Data can be read from/written to the row buffer by a read/write command, therefore the total latency is only the read/write command latency.

2. *Row-conflict*: The row address of the memory access is different from the address of the opened row. The memory access needs a precharge, an activate and a read/write command sequentially. The total latency is the sum of all three command latencies.

3. *Row-closed*: There is no valid data in the row buffer (i.e. closed). The access needs an activate command and then a read/write command. The total latency is the sum of these two command latencies.

DRAM access time is shortest in the case of a row-hit [15][4]. Therefore, a memory controller can try to maximize DRAM data throughput by maximizing the hit rate in the row buffer, i.e. by first scheduling row-hit requests among all the requests in the memory request buffer. Previous work [27] introduced the commonly-employed FR-FCFS (First Ready-First Come First Serve) policy which

[4]Row-hit latency is about one third of the latency of a row-conflict for contemporary SDRAM systems. For example, the row-hit and row-conflict latencies are 12.5ns and 37.5ns respectively for a 2Gbit DDR3 SDRAM chip [15] The row-closed latency is 25ns. We use the open-row policy throughout the paper since it increases the possibility to improve DRAM throughput. We also evaluate the closed-row policy in Section 6.8 to show the effectiveness of our mechanism.

prioritizes requests such that it services 1) row-hit requests first and 2) all else being equal, older requests first. This policy was shown to provide the best average performance in systems that do not employ hardware prefetching [27, 14]. Unfortunately, this policy is not aware of the interaction and interference between demand and prefetch requests in the DRAM system, and therefore treats demand and prefetch requests equally.

2.2. Hardware Prefetchers

In most of our experiments we use a stream prefetcher similar to the one used in IBM's POWER 4/5 [34]. Stream prefetchers are commonly used in many processors [34, 9] since they do not require significant hardware cost and work well for a large number of applications. They try to identify sequential streams of data that the application needs by closely monitoring and recording previous sequential accesses. Once a stream is identified, prefetch requests are sent out for data further down the stream so that when the processor actually demands the data, it will already be in the cache. As such, stream prefetchers are likely to generate many useful row-hit prefetch requests which our PADC can take advantage of. Therefore, we achieve significant performance improvements with our PADC combined with a stream prefetcher. Our implementation of a stream prefetcher is explained in the next section.

We also evaluated our mechanism with other prefetchers. The stride prefetcher [1] is similar to the stream prefetcher but instead of identifying only sequential streams, it detects sequences of addresses that differ by a constant value (stride) and generates prefetch requests that continue in the stride pattern. The Markov Prefetcher [7], a correlation-based prefetcher, records in a large table the cache miss addresses that follow a cache miss. If the same miss occurs again, the table is accessed, and prefetch requests to the recorded next address(es) are generated. We also evaluated our PADC with the previously proposed, CZone/ Delta Correlation (C/DC) prefetcher [24]. This prefetcher divides the address space statically into fixed size regions (CZones) and finds patterns among miss addresses within a region. It uses delta correlations to detect patterns more complex than simple strides, and generates prefetch requests that follow the detected pattern. Note that all of these prefetchers can also generate a significant amount of row-hit prefetch requests, especially for streaming/striding address/correlation patterns. In the results section we show that our PADC can improve performance when combined with any of these prefetchers and is not restricted to working with only a stream prefetcher.

2.3. Stream Prefetcher

We discuss an implementation of an aggressive stream prefetcher (similar to the one used in the IBM POWER4/5 [34]) used to collect most of our experimental results[5]. The stream prefetcher prefetches cache lines into the L2 cache[6].

Each *stream entry* in the stream prefetcher monitors a range of consecutive cache line addresses beginning at a starting pointer (S) and ending at S plus a *prefetch distance* (D). We call this range of consecutive cache line addresses a monitoring region. We also associate with the stream prefetcher a *prefetch degree* (N). When a new (not part of an existing stream) cache miss occurs, a stream entry is allocated and the cache line address is recorded as S. When subsequent cache accesses (both cache hits and misses) within a small distance from S occur, the direction of prefetching is determined and the monitoring region is setup starting with S and ending with S+D. If an L2 cache access happens within the monitoring region, N consecutive prefetch requests from cache line address S+D+1 to S+D+N are sent to the memory system. After prefetches are sent, the monitoring region is shifted in the direction of prefetching by the number of the requests sent. The timeliness and aggressiveness of the stream prefetcher are functions of both D and N.

Since the stream prefetcher tries to fetch consecutive cache lines from a region, it is likely that it will generate many row-hit

[5]The stream prefetcher we use for the evaluations is the best performing among the large set of prefetchers we examined. It improves IPC performance by 20% on average for all the 55 SPEC 2000/2006 benchmarks using demand-first policy. For the detail implementation of the stream prefetcher, refer to [32, 34].

[6]We implement a prefetcher that prefetches data into the L2 cache (the last-level cache in our processor model) only, since a large instruction window in an out-of-order processor can tolerate the latency of L1 misses most of time.

prefetch requests. If the row buffer locality of these prefetch requests is not exploited intelligently by the DRAM scheduling policy, DRAM throughput and system performance can degrade.

3. Motivation: Rigid Prefetch Scheduling in DRAM Systems

None of the existing DRAM scheduling policies [7, 27, 37, 11, 31, 32] take into account both the non-uniform nature of DRAM access latencies and the behavior of prefetch requests, i.e., whether they are useful or useless. We illustrate why a rigid, non-adaptive prefetch scheduling policy degrades performance in Figure 2. Consider the example in Figure 2(a), which shows three outstanding memory requests (to the same bank) in the memory request buffer. Row A is currently open in the row buffer of the bank. Two requests are prefetches (to addresses X and Z) that access row A while one request is a demand request (to address Y) that accesses row B.

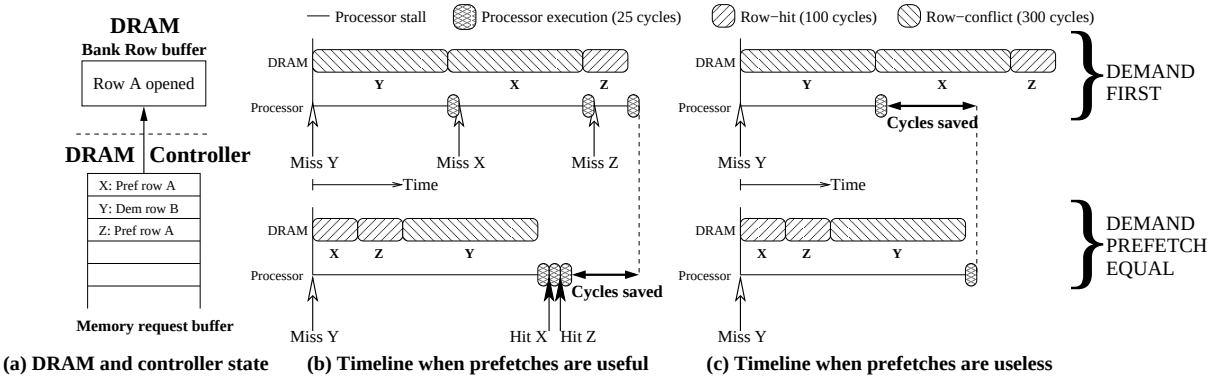


Figure 2. Performance impact of demand-first vs. demand-prefetch-equal policy

For Figure 2(b), assume that the processor needs to load addresses Y, X, and Z in a serial fashion (i.e. the prefetches are useful) and the computation between each load instruction takes a fixed, small number of cycles (25 in the figure) that is significantly smaller than the DRAM access latency[7]. We assume processor execution takes a small number of cycles because previous studies [21, 8] have shown that most of the execution time is dominated by DRAM access latency. Figure 2(b) shows the service timeline of the requests in DRAM and the resulting execution timeline of the processor for two different memory scheduling policies, *demand-first* and *demand-prefetch-equal*. With the demand-first policy (top), the row-conflict demand request is satisfied first, which causes the prefetch of address X to incur a row-conflict as well. The subsequent prefetch request to Z is a row-hit because the prefetch of X opens row A. As a result, the processor first stalls for approximately two row-conflict latencies (except for a small period of execution), to access address Y and then to access address X. The processor then stalls for an additional row-hit latency (again with the exception of another small period of execution) since it requires the data prefetched from address Z. The total execution time is the sum of two row-conflict latencies and one row-hit latency plus a small period of processor execution (the other computation periods are hidden), which is 725 cycles in the example.

With the demand-prefetch-equal policy (bottom), the row-hit prefetch requests to X and Z are satisfied first. Then, the row-conflict demand request to Y is serviced. The processor stalls until the demand request to Y is serviced. However, once the demand request is serviced, the processor does not stall any more because the memory requests to the other addresses it needs (X and Z) have already been serviced and placed into the cache. The processor only needs to perform the computations between the load instructions, and finds that loads to X and Z hit in the cache. The resulting total execution time is the sum of two row-hit latencies,

[7]For simplicity of illustration, this example abstracts away many details of the DRAM system as well as processor queues such as DRAM bus/bank timing constraints and processor queuing delays. These effects are faithfully modeled in our simulation framework. We omit them from the figure to illustrate the concept of rigid prefetch scheduling in the DRAM controller.

one row-conflict latency, and the latency to execute the computation between each load instruction for a total of only 575 cycles in the example. Hence, **treating prefetches and demands equally can significantly improve performance when prefetch requests are useful**. We observe that the stream prefetcher generates very accurate prefetch requests for many memory intensive applications such as *libquantum*, *swim*, and *leslie3d*. For these applications, the demand-prefetch-equal memory scheduling policy increases prefetch timeliness by increasing DRAM throughput and therefore improves performance significantly as shown in Figure 1.

However, prefetch requests might not always be useful. In the example of Figure 2(a), assume that the processor needs to load only address Y but still generates useless prefetches to addresses X and Z. Figure 2(c) shows the service timeline of the requests and the resulting execution timeline of the processor for the two different memory scheduling policies. With the demand-first policy (top), the processor needs to stall only for a single row-conflict latency that is required to service the demand request to Y, and therefore the total execution time is 325 cycles. On the other hand, with the demand-prefetch-equal policy, the processor needs to stall additional cycles since X and Z are serviced (even though they are not needed) before Y. It takes two row-hit requests to service the useless prefetches to X and Z and one row-conflict request to service the demand request to Y. The resulting execution time is 525 cycles. Hence, **treating prefetches and demands equally can significantly degrade performance when prefetch requests are useless**. In fact, our experimental data in Figure 1 showed that treating demands and prefetches equally in applications where most of the prefetches are useless causes prefetching to degrade performance by up to 36% (for *milc*).

These observations illustrate that 1) DRAM scheduling policies that rigidly prioritize between demand and prefetch requests without taking into account the usefulness of prefetch requests can either degrade performance or fail to provide the best possible performance, and 2) the effectiveness of a particular prefetch prioritization mechanism significantly depends on the usefulness of prefetch requests. Based on these observations, to improve the effectiveness of prefetching we aim to develop an adaptive DRAM scheduling policy that dynamically changes the prioritization order of demands and prefetches by taking into account the usefulness of prefetch requests.

4. Mechanism: Prefetch-Aware DRAM Controller

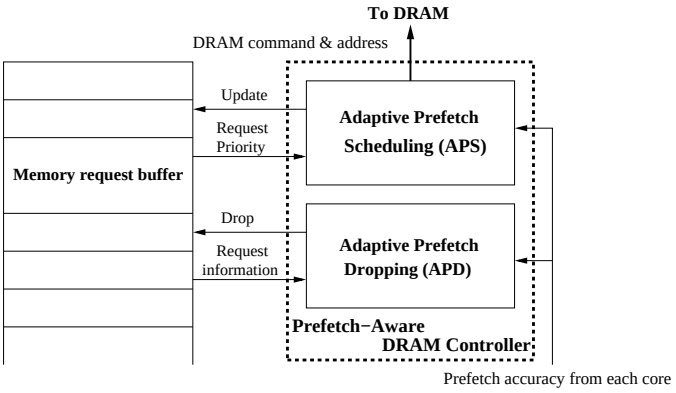


Figure 3. Prefetch-Aware DRAM Controller

Our Prefetch-Aware DRAM Controller (PADC) consists of two components as shown in Figure 3: An Adaptive Prefetch Scheduling (APS) unit and an Adaptive Prefetch Dropping (APD) unit. APS tries to 1) maximize the benefits of useful prefetches by increasing DRAM throughput and 2) minimize the harm of useless prefetches by delaying their DRAM service and hence reducing their interference with demand and useful prefetch requests. APD cancels useless prefetch requests effectively while preserving the benefits of useful prefetches. Both APS and APD are driven by the measurement of the prefetch accuracy of each processing core in a multi-core system. Before explaining how each component works, we explain how prefetch accuracy is measured for each core.

4.1. Prefetch Accuracy Measurement

We measure the prefetch accuracy for an application running on a particular core over a certain time interval. The accuracy is reset once the interval has elapsed so that the mechanism can adapt to the phase behavior of prefetching. To measure the prefetch accuracy of each core, the following hardware support is required:

1. Prefetch (P) bit per L2 cache line and memory request buffer entry[8]: For memory request buffer entries, this bit indicates whether or not the request was generated by the prefetcher. It is set when a new memory request is generated by the prefetcher, and reset when the processor issues a demand request to the same cache line while the prefetch request is still in the memory request buffer. For cache lines, this bit indicates whether or not a cache line was brought into the cache by a prefetch request. It is set when the line is filled (only if the prefetch bit of the request is set) and is reset when a cache hit to the same line occurs.

2. Prefetch Sent Counter (PSC) per core: This counter keeps track of the total number of prefetch requests sent by core. It is incremented whenever a prefetch request is sent to the memory request buffer by the core.

3. Prefetch Used Counter (PUC) per core: This counter keeps track of the number of prefetches that are useful. It is incremented when a prefetched cache line is used (cache hit) by a demand request and also when a demand request matches a prefetch request already in the memory request buffer.

4. Prefetch Accuracy Register (PAR) per core: This register stores the prefetch accuracy measured every time interval. PAR is computed by dividing PUC by PSC.

At the end of every time interval, PAR is updated with the prefetch accuracy calculated during that interval and PSC and PUC are reset to 0 to calculate the accuracy for the next interval. The PAR values for each core are fed into the Prefetch-Aware DRAM Controller which then uses the values to guide its scheduling and memory request buffer management policies.

4.2. Adaptive Prefetch Scheduling

Adaptive Prefetch Scheduling (APS) changes the priority of demand/prefetch requests from a processing core based on the prefetch accuracy estimated for that core. The basic idea is to 1) treat useful prefetch requests the same as demand requests so that useful prefetches can be serviced faster by maximizing DRAM throughput, and 2) give demand and useful prefetch requests a higher priority than useless prefetch requests so that useless prefetch requests do not interfere with demand and useful prefetch requests.

If the prefetch accuracy is greater than or equal to a certain threshold, $promotion\_threshold$, all of the prefetch requests from that core increase in priority and are treated the same as demand requests. Such prefetch requests and all demand requests are called *critical* requests. If the estimated prefetch accuracy of a core is less than $promotion\_threshold$, than demand requests of that core are prioritized over prefetch requests. Such prefetch requests are called *non-critical* requests.

The essence of our proposal is to prioritize critical requests over non-critical ones in the memory controller, while preserving DRAM throughput. To accomplish this, our mechanism prioritizes memory requests in the order shown in Rule 1. Each prioritization decision in this set of rules is described in further detail below.

Rule 1 Adaptive Prefetch Scheduling

1. **Critical request (C)**: Critical requests are prioritized over all other requests.
2. **Row-hit request (RH)**: Row-hit requests are prioritized over row-conflict requests.
3. **Urgent request (U)**: Demand requests generated by cores with low prefetch accuracy are prioritized over other requests.
4. **Oldest request (FCFS)**: Older requests are prioritized over younger requests.

[8] Many previous proposals [4, 28, 40, 41, 32] already use a prefetch bit for each cache line and memory request buffer entry.

First, critical requests (useful prefetches and demand requests) are prioritized over others. This delays the scheduling of non-critical requests, most of which are likely to be useless prefetches. As a result, useless prefetches are prevented from interfering with demands and useful prefetches.

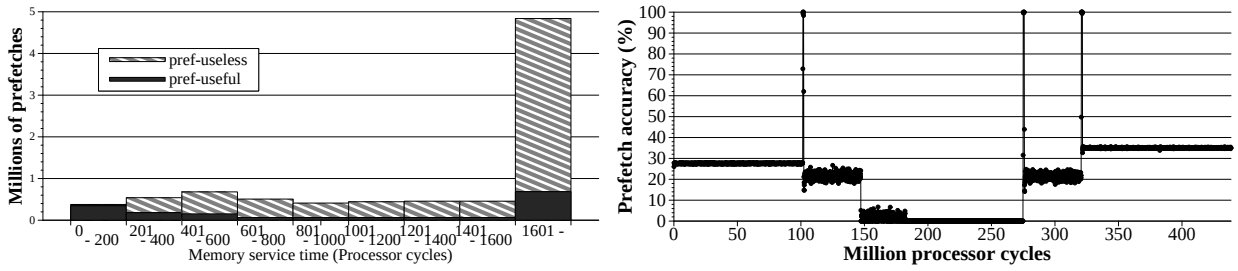
Second, row-hit requests are prioritized over others. This increases the row-buffer locality for demand and useful prefetch requests and maximizes DRAM throughput as much as possible.

Third, demand requests from cores whose prefetch accuracy is less than $promotion\_threshold$ are prioritized. These requests are called *urgent* requests. Intuitively, this rule tries to give a boost to the demand requests of a core with low prefetch accuracy over the demand requests of cores with high prefetch accuracy. This is due to two reasons. First, if a core has high prefetch accuracy, its prefetch requests will be treated the same as the demand requests of another core with low prefetch accuracy (due to the critical request first prioritization rule). Doing so risks starving the demand requests of the core with low prefetch accuracy, resulting in a performance degradation since many critical requests from the core with high prefetch accuracy (demand + prefetch requests) will contend with the critical requests from the core with low prefetch accuracy (demand requests only). To avoid such starvation and performance degradation, we boost the demand requests of the core with low prefetch accuracy. Second, the performance of a core with low prefetch accuracy is already affected negatively by the useless prefetches. By prioritizing the demand requests of such cores over the requests of other cores, we aim to help the performance of cores that are already losing performance due to poor prefetcher behavior. We further discuss the effect of prioritizing urgent requests in Section 6.3.4.

Finally, if all else is equal, older requests have priority over younger requests.

4.3. Adaptive Prefetch Dropping

APS naturally delays (just like the demand-first policy) the DRAM service of prefetch requests from applications with low prefetch accuracy by making the prefetch requests non-critical as described in Section 4.2. Even though this reduces the interference of useless requests with useful requests, it cannot get rid of all of the negative effects of useless prefetch requests (bandwidth consumption, cache pollution) because such requests will eventually be serviced. As such, APS by itself cannot eliminate all of the negative aspects of useless prefetches. Our second scheme, Adaptive Prefetch Dropping (APD), aims to overcome this limitation by proactively removing old prefetch requests from the request buffer if they have been outstanding for a long period of time. The key insight is that if a prefetch request is old (i.e., has been outstanding for a long time), it is likely to be useless and dropping it from the memory request buffer eliminates the negative effects the useless request might cause in the future. We first describe why old prefetch requests are likely to be useless based on empirical measurements.



(a) Prefetch service time with demand-first policy

(b) Prefetch accuracy every 100K cycles

Figure 4. Example of behavior of prefetches for *milc*

Why are old prefetch requests likely to be useless? Figure 4(a) shows the memory service time (from entry into the memory request buffer to entry into the L2 fill buffer) of both useful and useless prefetches for *milc* using the demand-first scheduling policy (other benchmarks show very similar behavior). The graph is a histogram with 9 latency intervals measured in processor cycles.

Each bar indicates the number of useful/useless prefetch requests whose memory service time was within that interval. 56% of all prefetches have a service time greater than 1600 processor cycles, and 86% of these prefetches are useless. Useful prefetches tend to have a shorter service time than useless prefetches (1486 cycles compared to 2238 cycles for *milc*). This is because a prefetch request that is waiting in the request buffer can become a demand request[9] if the processor sends a demand request for that same address while the prefetch request is still in the buffer. Such useful prefetches that are hit by demand requests will be serviced earlier by the demand-first prioritization policy. Therefore, useful prefetches on average experience a shorter service time than useless prefetches. This is also true when we apply APS since it prioritizes critical requests over non-critical requests.

Mechanism: The observation that old prefetch requests are likely to be useless motivates us to remove a prefetch request from the request buffer if the prefetch is old enough. Our proposal, APD, monitors prefetch requests for each core and invalidates any prefetch request that has been outstanding in the memory request buffer for longer than $drop\_threshold$ cycles. We adjust $drop\_threshold$ based on the prefetch accuracy for each core measured in the previous time interval. If the prefetch accuracy in the interval is low, our mechanism uses a relatively low value for $drop\_threshold$ so that it can quickly remove useless prefetches from the request buffer. If the prefetch accuracy is high in the interval, our mechanism uses a relatively high value for $drop\_threshold$ so that it does not prematurely remove useful prefetches from the request buffer. By removing useless prefetches, APD saves resources such as request buffer entries, DRAM bandwidth, and cache space, which can instead be used for critical requests (i.e. demand and useful prefetch requests) rather than being wasted on useless prefetch requests. Note that APD interacts positively with APS since APS naturally delays the service of useless (non-critical) requests so that the APD unit can completely remove them from the memory system thereby freeing up request buffer entries and avoiding unnecessary bandwidth consumption.

Determining $drop\_threshold$: Figure 4(b) shows the runtime behavior of the stream prefetcher accuracy for *milc*, an application that suffers from many useless prefetches. Prefetch accuracy was measured as described in Section 4.1 using an interval of 100K cycles. The figure clearly shows that prefetch accuracy can have very strong phase behavior. From 150 million to 275 million cycles, the prefetch accuracy is very low (close to 0%), implying many useless prefetch requests were generated during this time. Since almost all prefetches are useless during this period, we would like to be able to quickly drop them. Our mechanism accomplishes this using a low $drop\_threshold$. On the other hand, we would want $drop\_threshold$ to be much higher during periods of high prefetch accuracy. Our evaluation shows that a simple 4-level $drop\_threshold$ adjusted dynamically can effectively eliminate useless prefetch requests from the memory system while keeping useful prefetch requests in the memory request buffer.

4.4. Implementation and Hardware Cost of a Prefetch-Aware DRAM Controller

An implementation of our PADC requires storing additional information in each memory request buffer entry to support the priority and aging information needed by APS and APD. The required additional information (in terms of the fields added to each request buffer entry) is shown in Figure 5.

The C, RH and FCFS fields are already used in the baseline FR-FCFS demand-first policy to indicate criticality (demand/prefetch), row-hit status, and arrival time of the request. Therefore the only additional fields are U, P, ID, and AGE, which indicate the urgency, prefetch status, core ID, and age of the request. Each DRAM cycle, priority encoder logic chooses the highest priority request using the priority fields (C, RH, U, and FCFS) in the order shown in Figure 5.

The APD unit removes a prefetch request from the memory request buffer if the request is older than the $drop\_threshold$ of the core that generated the request. It does not remove a prefetch request (that is not scheduled for the DRAM service) until it ensures that the prefetch cannot be matched by a demand request. This is accomplished by invalidating the MSHR entry of the prefetch

[9] A prefetch request that is hit by a demand request in the memory request buffer becomes a real demand request. However, we count it as a useful prefetch throughout the paper since it was first requested by the prefetcher rather than the processing core.

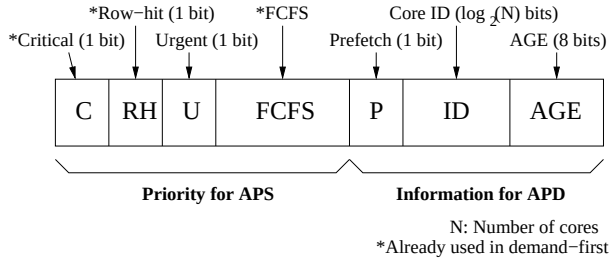


Figure 5. Memory request field for PADC

request before actually dropping it. The APD unit knows if a request is a prefetch and also which core it belongs to from the P and ID fields. The AGE field of each request entry keeps track of the age of the request. APD compares the AGE of the request to the corresponding core's $drop\_threshold$ and removes the request accordingly. Note that the estimation of the age of a request does not need to be highly accurate. For example, the AGE field is incremented every 100 processor cycles for our evaluation.

The hardware storage cost required for our implementation of the PADC is shown in Table 1. The storage cost for our 4-core CMP system described in Section 5 is shown in Table 2. The total storage cost is only 34,720 bits (∼4.25KB) which is equivalent to only 0.2% of the L2 cache data storage in our baseline processor. Note that the Prefetch bit (P) per cache line accounts for over 4KB of storage by itself (∼95% of the total required storage). If a processor already employs prefetch bits in its cache, the total storage cost of our prefetch-aware DRAM controller is only 1,824 bits (∼228B).

	Bit field	Count	Cost (bits)
Prefetch accuracy	P (1 bit)	$N_{cache} \times N_{core} + N_{req}$	$N_{cache} \times N_{core} + N_{req}$
	PSC (16 bits)	N_{core}	$N_{core} \times 16$
	PUC (16 bits)	N_{core}	$N_{core} \times 16$
	PAR (8 bits)	N_{core}	$N_{core} \times 8$
APS	U (1 bit)	N_{req}	N_{req}
APD	ID ($log_2 N_{core}$ bits)	N_{req}	$N_{req} \times log_2 N_{core}$
	AGE (10 bits)	N_{req}	$N_{req} \times 10$

Table 1. Hardware Cost of Prefetch-Aware DRAM Controller (N_{cache}: Number of cache lines per core N_{core}: Number of cores, N_{req}: Number of memory request buffer entries)

	Bit field	Cost (bits)
Prefetch accuracy	P	32,896
	PSC	64
	PUC	64
	PAR	32
APS	U	128
APD	ID	256
	AGE	1,280
Total storage cost for the 4-core system in Section 5		34,720
Total storage cost as a fraction of the L2 cache capacity		0.2%

Table 2. Hardware cost of PADC on 4-core system

5. Methodology

5.1. Processor Model and Workloads

We use a cycle accurate x86 CMP simulator for our evaluation. Our processor faithfully models port contention, queuing effects, bank conflicts, and other DDR3 DRAM system constraints. The baseline configuration of each processing core is shown in Table 3 and the shared resource configuration for single, 2, 4, and 8-core CMPs is shown in Table 4.

Execution Core	Out of order, 15 (fetch, decode, rename stages) stages, decode/retire up to 4 instructions, issue/execute up to 8 microinstructions; 256-entry reorder buffer; 32-entry load-store queue; 256 physical registers
Front End	Fetch up to 2 branches; 4K-entry BTB; 64-entry return address stack; Hybrid branch predictor: 64K-entry gshare [13] and 64K-entry PAs predictor [36] with 64K-entry selector
On-chip Caches	L1 I-cache: 32KB, 4-way, 2-cycle, 1 read port, 1 write port, 64B line size; L1 D-cache: 32KB, 4-way, 4-bank, 2-cycle, 1 read port, 1 write port, 64B line size; Unified L2: 512KB (1MB for single core processor), 8-way, 8-bank, 15-cycle, 1 port, 64B line size;
Prefetcher	Stream prefetcher with 32 streams, prefetch degree of 4, and cache line prefetch distance of 64 (lookahead) [34, 32]

Table 3. Baseline configuration per core

DRAM controller	On-chip, demand-first FR-FCFS scheduling policy; 1 memory controller for 1, 2, 4, 8-core CMP 64, 64, 128, 256-entry L2 MSHR and memory request buffer for 1, 2, 4, 8-core CMP.
DRAM and bus	667MHz DRAM bus cycle, Double Data Rate (DDR3 1333MHz) [15], 16B-wide data bus per memory controller 8 DRAM banks, 4KB row buffer per bank Latency: 15ns per command (precharge(^tRP), activate(^tRCD), read/write(CL)), BL = 4; AL = 3

Table 4. Baseline shared resource configuration

We use the SPEC 2000/2006 benchmarks for experimental evaluation. Each benchmark was compiled using ICC (Intel C Compiler) or IFORT (Intel Fortran Compiler) with the -O3 option. We ran each benchmark with the reference input set for 200 million x86 instructions selected by Pinpoints [25] as a representative portion of each benchmark.

We classify the benchmarks into three categories: prefetch-insensitive, prefetch-friendly, and prefetch-unfriendly (class 0, 1, and 2 respectively) based on the performance impact a prefetcher has on the application[10]. The characteristics for a subset of benchmarks with and without a stream prefetcher are shown in Table 5. Only a subset of benchmarks were chosen due to limited space, however, we do evaluate the entire set of 55 benchmarks for single core experiments for our results. To evaluate our mechanism on CMP systems, we formed combinations of multiprogrammed workloads from the 55 SPEC 2000/2006 benchmarks. We ran 54, 32, and 21 randomly chosen workload combinations (from the 55 SPEC benchmarks) for our 2, 4, and 8-core CMP configurations respectively.

	No prefetcher		Prefetcher with demand-first policy							No prefetcher		Prefetcher with demand-first policy					
Benchmark	IPC	MPKI	IPC	MPKI	RBH(%)	ACC(%)	COV(%)	Class	Benchmark	IPC	MPKI	IPC	MPKI	RBH(%)	ACC(%)	COV(%)	Class
eon_00	2.08	0.01	2.08	0.00	84.93	37.37	52.64	0	swim_00	0.35	27.57	0.62	8.66	42.83	99.95	68.58	1
mgrid_00	0.65	6.50	0.85	0.30	59.56	97.46	95.37	1	galgel_00	1.42	4.26	1.10	7.56	65.50	30.96	23.94	2
art_00	0.18	89.39	0.18	65.52	91.46	35.88	34.00	2	equake_00	0.42	19.87	1.00	0.76	85.02	95.63	96.19	1
facerec_00	1.40	3.45	1.64	1.18	92.42	55.15	67.04	1	ammp_00	1.70	0.80	1.47	1.70	56.20	5.96	8.03	2
lucas_00	0.48	10.61	0.79	1.42	44.06	86.78	86.63	1	gcc_06	0.55	6.28	0.81	2.23	81.57	32.62	65.37	1
mcf_06	0.13	33.73	0.15	29.70	25.63	31.43	14.75	1	hmmer_06	1.34	1.76	1.35	0.03	27.44	95.42	98.21	0
sjeng_06	1.57	0.38	1.57	0.38	25.13	1.67	1.11	0	omnetpp_06	0.41	10.16	0.44	9.57	61.86	10.50	18.33	2
libquantum_06	0.41	13.51	0.65	2.75	81.39	99.98	79.63	1	astar_06	0.43	10.19	0.48	9.23	43.86	18.38	12.64	1
xalancbmk_06	0.80	1.70	0.71	2.12	49.35	8.96	13.26	2	bwaves_06	0.59	18.71	1.23	0.37	83.99	99.97	98.00	1
gamess_06	2.11	0.04	2.13	0.01	83.97	57.80	74.73	0	milc_06	0.41	29.33	0.46	20.88	81.13	19.45	28.81	2
zeusmp_06	0.75	4.55	0.86	2.32	46.91	55.93	50.45	1	cactusADM_06	0.71	4.54	0.84	2.21	33.56	45.12	51.47	1
leslie3d_06	0.53	20.89	0.86	2.41	77.32	89.72	88.66	1	soplex_06	0.35	21.25	0.72	3.61	78.81	80.12	83.08	1
GemsFDTD_06	0.44	15.61	0.80	2.02	55.82	90.71	87.12	1	lbm_06	0.46	20.16	0.70	2.93	58.24	94.27	85.45	1
wrf_06	0.62	8.10	1.03	0.60	67.14	95.23	92.63	1	sphinx3_06	0.36	12.94	0.64	2.24	83.94	54.91	82.96	1

Table 5. Characteristics for 28 SPEC 2000/2006 benchmarks with/without stream prefetcher: IPC, MPKI (L2 misses Per 1K Instructions), RBH (Row Buffer Hit rate), ACC (prefetch accuracy), COV (prefetch coverage), and class

For the evaluation of our PADC, we use a prefetch accuracy value of 85% for $promotion\_threshold$ (for APS) and a dynamic threshold shown in Table 6 for $drop\_threshold$ (for APD). The accuracy is calculated every 100K cycles.

[10] If MPKI (L2 Misses Per 1K Instructions) increases when the prefetcher is enabled, the benchmark is classified as 2. If MPKI without prefetching is greater than 10 and bus traffic increases by more than 75% when prefetching is enabled the benchmark is also classified as 2. Otherwise, if IPC increases by 5%, the benchmark is classified as 1. Otherwise, it is classified as 0. Note that memory intensive applications that experience increased IPC and reduced MPKI (such as *milc*) may still be classified as prefetch-unfriendly if bus traffic increases significantly. The reason for this is that although an increase in bus traffic may not have much of a performance impact on single core systems, in CMP systems with shared resources, the additional bus traffic can degrade performance substantially.

Prefetch accuracy (%)	0 - 10	10 - 30	30 - 70	70 - 100
$drop\_threshold$ (processor cycles)	100	1,500	50,000	100,000

Table 6. Dynamic $drop\_threshold$ values for Adaptive Prefetch Dropping based on prefetch accuracy

5.2. Metrics

We define the metrics used for experimental evaluation in this section. *Bus traffic* is the number of cache lines transferred over the bus during the execution of a workload. It comprises the cache lines brought in from demand, useful prefetch, and useless prefetch requests. We define *Prefetch accuracy (ACC)* and *coverage (COV)* as follows:

$$ACC = \frac{Number\ of\ useful\ prefetches}{Number\ of\ prefetches\ sent}, \quad COV = \frac{Number\ of\ useful\ prefetches}{Number\ of\ demand\ requests + Number\ of\ useful\ prefetches}$$

To evaluate the effect of DRAM throughput improvement on the processing core, we define *instruction window Stall cycles Per Load instruction (SPL)* which indicates on average how much time the processor spends idly waiting for DRAM service.

$$SPL = \frac{Total\ number\ of\ window\ stall\ cycles}{Total\ number\ of\ load\ instructions}$$

To measure CMP system performance, we use *Individual Speedup (IS)*, *Weighted Speedup (WS)* [30], and *Harmonic mean of Speedups (HS)* [12]. As shown by Eyerman and Eeckhout [3], WS corresponds to system throughput and HS corresponds to the inverse of job turnaround time. In the equations that follow, N is the number of cores in the CMP system. IPC^{alone} is the IPC measured when an application runs alone on one core in the CMP system (other cores are idle) and $IPC^{together}$ is the IPC measured when an application runs on one core while other applications are running on the other cores of a CMP. Unless otherwise mentioned, we use the demand-first policy to measure IPC^{alone} for all of our experiments to show the effectiveness of our mechanism on CMP systems.

$$IS_i = \frac{IPC_i^{together}}{IPC_i^{alone}}, \quad WS = \sum_i^N \frac{IPC_i^{together}}{IPC_i^{alone}}, \quad HS = \frac{N}{\sum_i^N \frac{IPC_i^{alone}}{IPC_i^{together}}}$$

6. Experimental Evaluation

6.1. Single-Core Results

Figure 6 shows the performance of PADC on a single core system. IPC is normalized to the baseline which employs the demand-first scheduling policy. We show the performance of only 15 individual benchmarks due to limited space. The rightmost bars show the average performance of all 55 benchmarks (*gmean55*). As discussed earlier, neither of the rigid scheduling policies (demand-first, demand-prefetch-equal) provides the best performance across all applications. Demand-first performs better for most prefetch-unfriendly benchmarks (class 2) such as *galgel*, *art* and *ammp* while demand-prefetch-equal does better for most prefetch-friendly ones (class 1) such as *swim*, *libquantum* and *lbm*. Averaged over all 55 benchmarks, the demand-prefetch-equal policy outperforms demand-first by 0.5% since there are more benchmarks (29 out of 55) that belong to class 1.

Adaptive Prefetch Scheduling (APS), shown in the fourth bar from the left, effectively adapts to the behavior of the prefetcher. In most benchmarks, APS provides at least as good performance as the best rigid prefetch scheduling policy. As a result, APS improves performance by 3.6% over all 55 benchmarks compared to the baseline. APS (and demand-prefetch-equal) improves performance over demand-first for many prefetch friendly applications such as *libquantum*, *bwaves*, and *leslie3d*. This is due to two reasons.

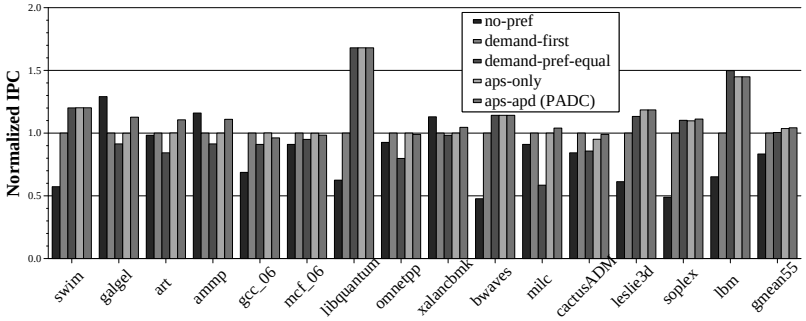


Figure 6. Performance for 55 SPEC benchmarks on single core system: Normalized IPC for 15 benchmarks and average for all 55 (gmean55)

First, APS increases DRAM throughput in these applications because it treats demands and prefetches equally most of the time. Doing so improves the timeliness of the prefetcher because prefetch requests do not get delayed behind demand requests. Second, improved DRAM throughput reduces the probability of the memory request buffer being full. As a result, more prefetches are able to enter the request buffer. This improves the coverage of the prefetcher as more useful prefetch request get a chance to be issued. For example, APS improves the prefetch coverage from 80%, 98%, and 89% to 100%, 100%, and 92% for *libquantum*, *bwaves*, and *leslie3d* respectively (shown in Figure 8(a)).

On the other hand, even though APS is able to provide the performance of the best rigid prefetch scheduling policy for each application, it is unable to overcome the performance loss due to prefetching in some prefetch-unfriendly applications, such as *galgel*, *ammp* and *xalancbmk*. The prefetcher generates many useless prefetches in these benchmarks that a simple DRAM scheduling policy cannot eliminate. Incorporating adaptive prefetch dropping (APD) in addition to APS significantly improves performance especially in prefetch-unfriendly applications. Using APD recovers part of the performance loss due to prefetching in *galgel*, *ammp*, and *xalancbmk* because it eliminates 54%, 76%, and 54% of the useless prefetch requests respectively (shown in Figure 8(a)). As a result, using both of our proposed mechanisms (APD in conjunction with APS) provides 4.3% performance improvement over the baseline.

Figure 7 provides insight into the performance improvement of the proposed mechanisms by showing the effect of each mechanism on the stall time experienced per load instruction (SPL). Our PADC reduces SPL by 5.0% compared to the baseline. By providing better DRAM scheduling and eliminating useless prefetches, PADC reduces the amount of time the processor stalls for each load instruction and allows the processor to make faster progress. As a result, PADC significantly improves performance.

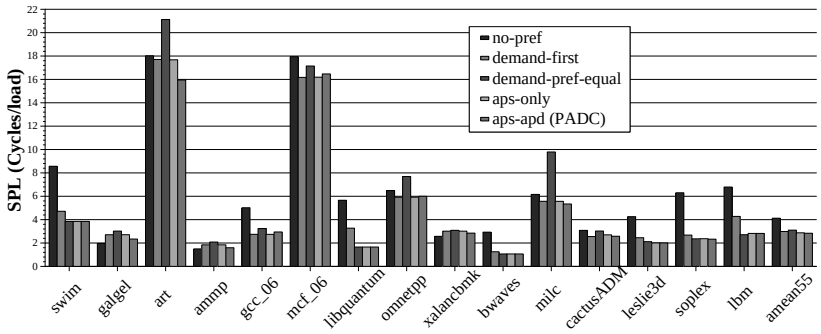


Figure 7. Stall time per load (SPL) on the single core system

Figure 8 breaks down the bus traffic into three categories: useful prefetches, useless prefetches, and demand requests. The PADC reduces bus traffic by 10.4% across all benchmarks (amean55) as shown. Reduction in bus traffic is mainly due to APD

which significantly reduces the number of useless prefetches. For many benchmarks, APS by itself provides the same bandwidth consumption provided by the best rigid policy for each benchmark. We conclude that our prefetch-aware DRAM controller is very effective at improving both performance and bandwidth-efficiency in single-core systems.

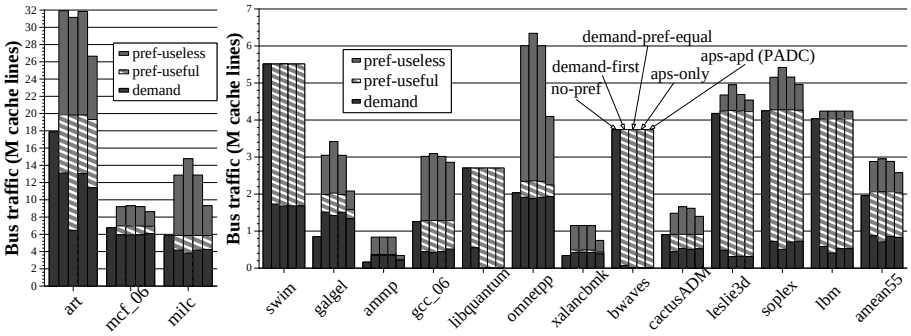


Figure 8. Bus traffic on single core system

6.1.1. Effect of PADC on Row Buffer Hit Rate Recall that the demand-prefetch-equal policy prioritizes row-hit requests regardless of whether a request is a prefetch or demand. If we consider all demand and prefetch requests (regardless of whether or not a prefetch is useful) for the entire run of an application, the demand-prefetch-equal policy will result in the highest row buffer hit rate (RBH) and therefore the lowest average DRAM access latency among all considered policies. However, this does not mean that this policy performs best since prefetches are NOT always useful as discussed in Section 6.1. When prefetching is enabled, we need a better metric to show how a mechanism reduces effective memory latency. Hereby, we define row buffer hit rate for useful (demand and useful prefetch) requests (RBHU) as follows:

$$RBHU = \frac{Number\ of\ row\text{-}hit\ demand\ requests + Number\ of\ useful\ row\text{-}hit\ prefetch\ requests}{Number\ of\ demand\ requests + Number\ of\ useful\ prefetch\ requests}$$

The demand-prefetch-equal policy will still show the highest RBHU, since RBHU is also maximized by prioritizing row-hit requests. However, a good DRAM scheduling mechanism should keep its RBHU close to demand-prefetch-equal's RBHU because it should aim to maximize DRAM bandwidth for useful requests. Table 7 shows RBHU values for 13 benchmarks on the single core processor with no prefetching, demand-first, demand-prefetch-equal, APS, and PADC. The RBHU of APS is very close to that of demand-prefetch-equal and significantly better than the RBHU of demand-first since APS successfully exploits row buffer locality for useful requests.

Employing APD with APS (i.e. PADC) slightly reduces RBHU for some applications such as *galgel*, *ammp*, *mcf*, *omnetpp*, *xalancbmk*, and *soplex*. This is because adaptive prefetch dropping cancels some useful prefetches as shown in Figure 8, thereby reducing the fraction of useful row buffer hits. Nonetheless, APD improves overall performance for these applications since it reduces the contention between demands and prefetches by eliminating a significant number of useless prefetches as discussed in Section 6.1.

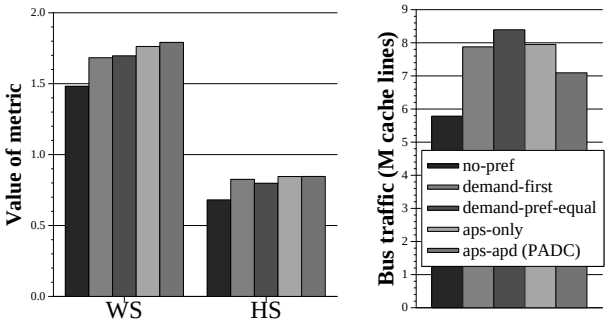
6.2. 2-Core Results

We briefly discuss only the average performance and bus traffic for the 54 workloads on the 2-core system due to space limitations (our main analysis focuses on 4-core systems). Figure 9 shows that PADC improves both performance metrics (WS and HS) by 8.4%, and 6.4% respectively compared to the demand-first policy and also reduces the memory bus traffic by 10.0%. Thus, the

	swim	galgel	art	ammp	mcf_06	libquantum	omnetpp	xalancbmk	bwaves	milc	leslie3d	soplex	lbm_06	amean55
no-pref	0.18	0.51	0.94	0.40	0.12	0.86	0.47	0.23	0.76	0.85	0.71	0.81	0.53	0.55
demand-first	0.44	0.58	0.94	0.48	0.19	0.86	0.56	0.27	0.87	0.88	0.81	0.87	0.64	0.63
demand-pref-equal	0.50	0.58	0.96	0.50	0.23	0.98	0.59	0.28	0.89	0.90	0.91	0.93	0.92	0.68
aps	0.50	0.58	0.94	0.48	0.19	0.98	0.56	0.27	0.89	0.88	0.90	0.91	0.90	0.66
aps-apd (PADC)	0.50	0.56	0.94	0.44	0.18	0.98	0.54	0.25	0.89	0.88	0.90	0.90	0.90	0.65

Table 7. Row buffer hit rate for useful (demand and useful prefetch) requests

proposed mechanism is effective for dual-core systems. We do not discuss these results further since dual-core processors are no longer the state-of-the-art in multi-core systems.



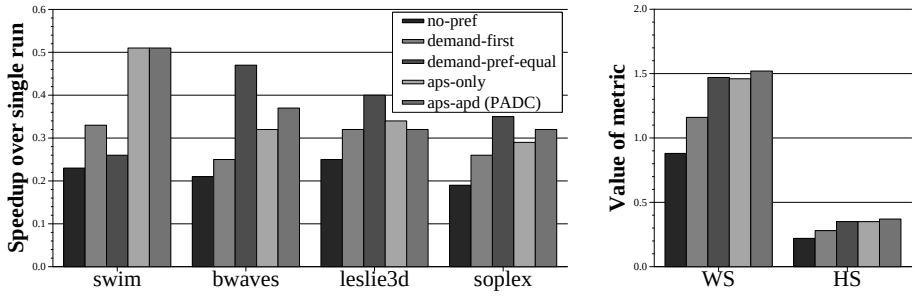
(a) Average system performance (b) Average traffic

Figure 9. Overall performance for 54 workloads on the 2-core system

6.3. 4-Core Results

We ran 32 different workloads to evaluate the effectiveness of PADC on the 4-core system. In the following sections, we discuss three cases in detail to provide insights into the behavior of Prefetch-Aware DRAM Controller.

6.3.1. Case Study I: All Prefetch-Friendly Applications Our first case study examines the behavior of our proposed mechanisms when four prefetch-friendly applications (*swim, bwaves, leslie3d,* and *soplex*) are run together on the 4-core system. Figure 10(a) shows the speedup of each application and Figure 10(b) shows system performance.



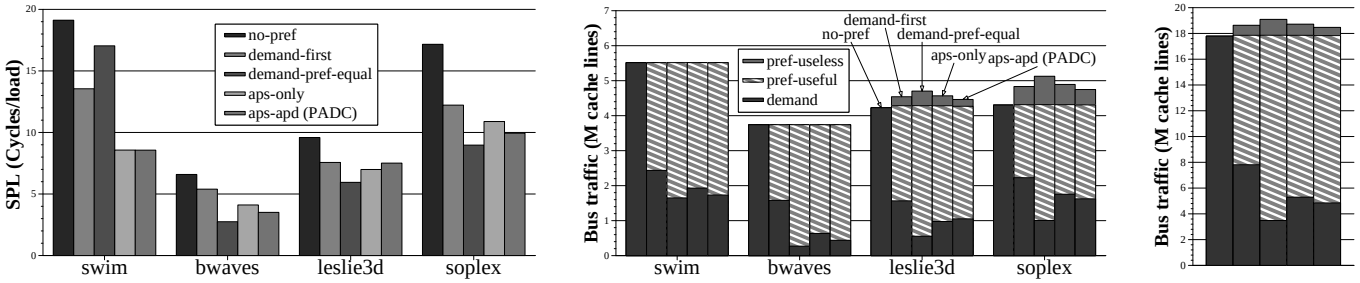
(a) Individual speedup (b) System performance

Figure 10. A prefetch-friendly 4-core workload: (a) Individual application speedup, (b) System performance

In addition, Figure 11 provides insight into the performance changes by showing how each mechanism affects stall-time per load as well as memory bus traffic. Several observations are in order:

First, since all 4 applications are prefetch-friendly (i.e., prefetcher has very high coverage as shown in Figure 11(b)), prefetching provides significant performance improvement in all applications regardless of the DRAM scheduling policy. In addition, the demand-prefetch-equal policy significantly outperforms demand-first policy (by 28% in terms of weighted speedup) because

prefetches are very accurate in all 4 applications. The demand-prefetch-equal policy reduces stall-time per load as shown in Figure 11(a) because it improves DRAM throughput.



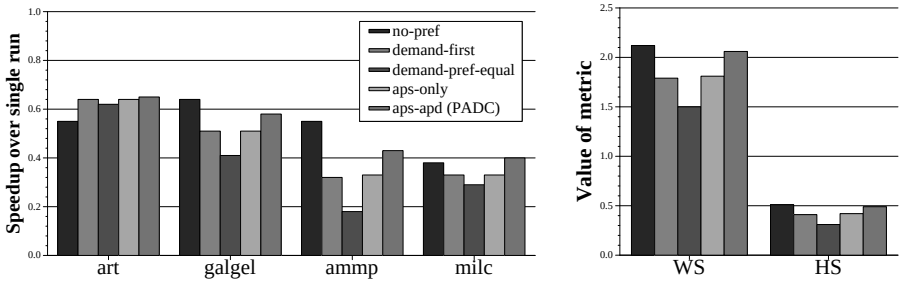
(a) SPL (b) Bus traffic breakdown (c) Total traffic

Figure 11. A prefetch-friendly 4-core workload: (a) SPL, (b) Bus traffic per application, (c) Total bus traffic

Second, our PADC outperforms both of the rigid prefetch scheduling policies improving weighted speedup by 31.3% over the baseline demand-first policy. This is because it 1) successfully prioritizes critical (useful) requests over others thereby reducing SPL, and 2) drops useless prefetches in *leslie3d* and *soplex* thereby reducing their negative effects on all applications. Consequently, our PADC also improves prefetch coverage from 56% to 73% (as shown in Figure 11(c)). This is because it improves DRAM throughput and reduces contention for memory system resources by dropping useless prefetches from *leslie3d* and *soplex* allowing more useful prefetches to enter the memory system.

Finally, the bandwidth savings provided by PADC is relatively small (0.9%) because these applications do not generate a large number of useless prefetch requests. However, there is still a non-negligible reduction in bus traffic due to the effective dropping of some useless prefetches in *leslie3d* and *soplex*. We conclude that Prefetch-Aware DRAM Controller can provide performance and bandwidth-efficiency improvements even when all applications benefit significantly from prefetching.

6.3.2. Case Study II: All Prefetch-Unfriendly Applications Our second case study examines the behavior of our proposed mechanisms when four prefetch-unfriendly applications (*art*, *galgel*, *ammp*, and *milc*) are run together on the 4-core system. Since the prefetcher is very inaccurate for all applications, prefetching degrades performance regardless of the scheduling policy. However, as shown in Figure 12, the demand-first policy and APS provide better performance than the demand-prefetch-equal policy by prioritizing demand requests over prefetch requests which are more than likely to be useless. Employing adaptive prefetch dropping drastically reduces the useless prefetches in all four applications (see Figure 13(b)), and therefore frees up memory system resources to be used by demands and useful prefetch requests. The effect of this can be seen by the reduced SPL (shown in Figure 13(a)) for all applications. As a result, our PADC performs better than the best previous prefetch scheduling policy for *all* applications.

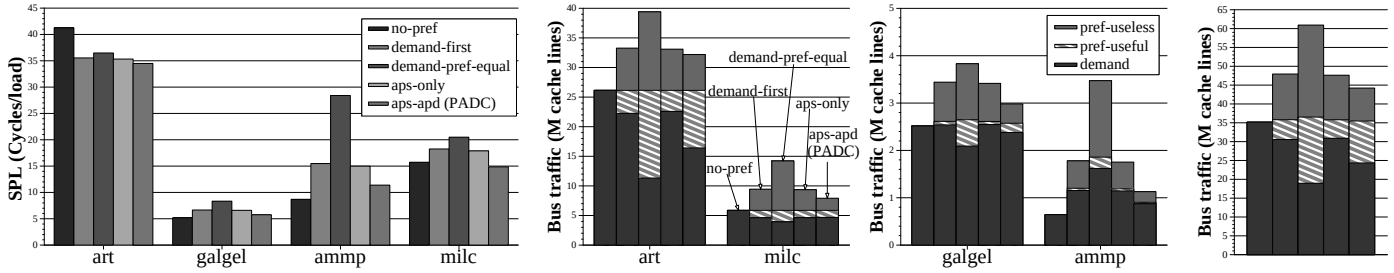


(a) Individual speedup (b) System performance

Figure 12. A prefetch-unfriendly 4-core workload: (a) Individual application speedup, (b) System performance

PADC improves system performance by 17.7% (weighted speedup) and 21.5% (harmonic mean of speedups), while reducing

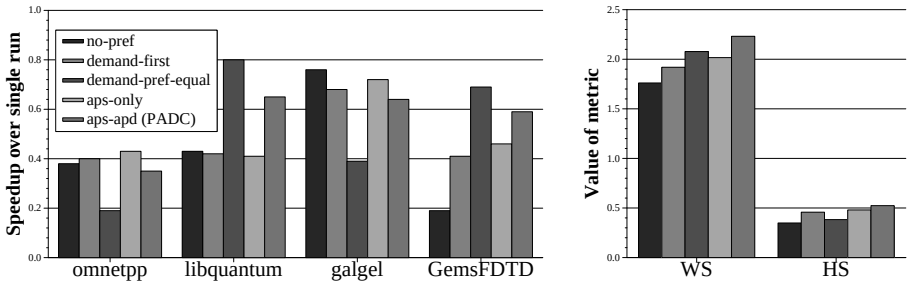
bandwidth consumption by 9.1% over the baseline demand-first scheduler as shown in Figure 13(c). By largely reducing the negative effects of useless prefetches both in scheduling and memory system buffers/resources, PADC almost eliminates the system performance loss observed in this prefetch-unfriendly mix of applications. Weighted speedup and harmonic mean of speedups obtained with our PADC is within 2% and 1% of those obtained with no prefetching. We conclude that Prefetch-Aware DRAM Controller can effectively eliminate the negative performance impact caused by inaccurate prefetching by intelligently managing the scheduling and buffer management of prefetch requests even in workload mixes where prefetching performs inefficiently for all applications.



(a) SPL (b) Bus traffic breakdown (c) Total traffic

Figure 13. A prefetch-unfriendly 4-core workload: (a) SPL, (b) Bus traffic per application, (c) Total bus traffic

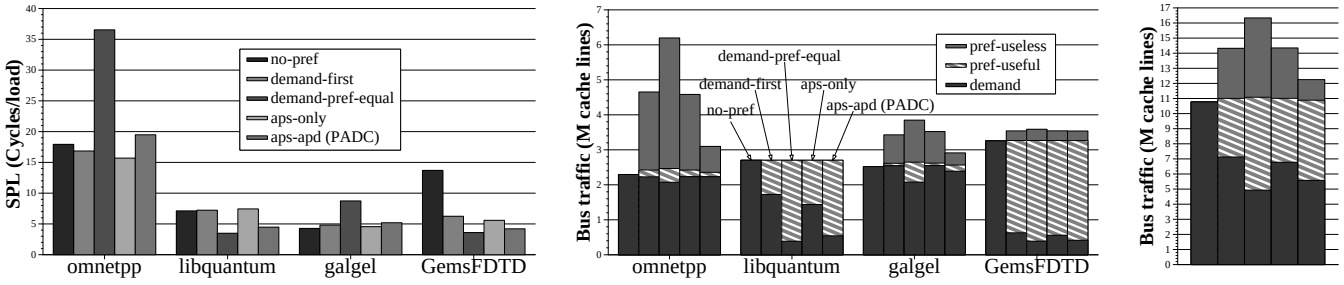
6.3.3. Case Study III: Mix of Prefetch-Friendly and Prefetch-Unfriendly Applications Figures 14 and 15 show performance and bus traffic when two prefetch-friendly (*libquantum* and *GemsFDTD*) and two prefetch-unfriendly (*omnetpp* and *galgel*) applications are run together. The prefetches for *libquantum* and *GemsFDTD* are very beneficial. Therefore demand-prefetch-equal significantly improves weighted speedup. However, the prefetcher generates many useless prefetches for *omnetpp* and *galgel* as shown in Figure 15(a). These useless prefetches temporarily deny service to critical requests from the two other cores. Because APD eliminates a large portion (67% and 57%) of all useless prefetches in *omnetpp* and *galgel*, it frees up both request buffer entries and bandwidth in the memory system. These freed up resources are utilized efficiently by the critical requests of *libquantum* and *GemsFDTD* thereby significantly improving their individual performance, while slightly reducing *omnetpp* and *galgel*'s individual performance. Since it eliminates a large amount of useless prefetches, PADC reduces total bandwidth consumption by 14.5% over the baseline demand-first policy. We conclude that PADC can effectively prevent the denial of service caused by the useless prefetches of prefetch-unfriendly applications on the useful requests of other applications.



(a) Individual speedup (b) System performance

Figure 14. A mixed 4-core workload: (a) Individual application speedup, (b) System performance

6.3.4. Effect of Prioritizing Urgent Requests In this section, we discuss the effectiveness of prioritizing urgent requests using the application mix in case study III. We say that a multi-core system is *fair* if each application experiences the same individual



(a) SPL (b) Bus traffic breakdown (c) Total traffic

Figure 15. A mixed 4-core workload: (a) SPL, (b) Bus traffic per application, (c) Total bus traffic

speedup when multiple applications run together on the system. To indicate the degree of unfairness, we define *Unfairness (UF)* [3] as follows:

$$UF = \frac{MAX(IS_0,\ IS_1,\ \ldots,\ IS_{n-1})}{MIN(IS_0,\ IS_1,\ \ldots,\ IS_{n-1})}, \quad N:\ Number\ of\ Cores$$

Table 8 shows individual speedup, unfairness, weighted speedup, and harmonic mean of speedups for the workload from case study III for five policies: demand-first, versions of APS and PADC that do not use the concept of "urgent requests," and regular APS and PADC (with "urgent requests"). If the concept of "urgent requests" is not used, demand requests from the prefetch-unfriendly applications (*omnetpp* and *galgel*) unfairly starve because a large number of critical requests from the prefetch-friendly applications (*libquantum* and *GemsFDTD*) are given the same priority as those demand requests. This starvation, combined with the negative effects of useless prefetches, leads to unacceptably low individual speedups for these applications, resulting in large unfairness. When urgency is used to prioritize requests, this unfairness is significantly mitigated, as shown in Table 8. In addition, harmonic mean of speedups (i.e., average job turnaround time) significantly improves at the cost of very little weighted speedup (i.e., system throughput) degradation. However, we found that for most workloads (30 out of the 32), prioritizing urgent requests improves weighted speedup as well. This trend holds true for most workload mixes that consist of prefetch-friendly and prefetch-unfriendly applications. On average (not shown in the table), prioritizing urgent requests improves UF, HS, and WS by 13.7%, 8.8%, and 3.8% respectively compared to PADC with no urgency concept for the 32 4-core workloads. We conclude that the concept of urgency significantly improves system fairness while keeping system performance high.

	Individual speedup				UF	WS	HS
	omnetpp	libquantum	galgel	GemsFDTD			
demand-first	0.40	0.42	0.68	0.41	1.69	1.92	0.46
aps-no-urgent	0.26	0.68	0.47	0.61	2.57	2.02	0.44
aps	0.43	0.41	0.72	0.46	1.73	2.02	0.48
aps-apd-no-urgent	0.21	0.94	0.42	0.70	4.55	2.26	0.41
aps-apd (PADC)	0.35	0.65	0.64	0.59	1.84	2.23	0.52

Table 8. Effect of prioritizing urgent requests

6.3.5. Effect on Identical-Application Workloads It is common that commercial servers frequently run multiple instances of identical applications. In this section, we evaluate the effectiveness of PADC when the 4-core system runs four identical applications together. Since APS prioritizes memory requests and APD drops useless prefetches, both based on the estimated prefetch accuracy, PADC should evenly improve individual speedup of each instance of the identical applications running together. In other words, all the instances of the application are likely to show the same behavior and the same adaptive decision should be made for every interval.

Table 9 shows the system performance of PADC when four instances of *libquantum* are run together on the 4-core system. Because *libquantum* is very prefetch-friendly and most prefetches are row-hits, the demand-prefetch-equal policy performs very well by achieving almost the same speedup for all four instances. APS and PADC perform similarly to the demand-prefetch-equal (improving weighted speedup by 18.2% compared to demand-first) since they successfully treat demands and prefetches equally for all four instances.

	Individual speedup				WS	HS	UF
	libquantum	libquantum	libquantum	libquantum			
no-pref	0.60	0.60	0.60	0.59	2.40	0.60	1.01
demand-first	0.69	0.67	0.65	0.64	2.66	0.66	1.08
demand-pref-equal	0.80	0.79	0.78	0.77	3.14	0.78	1.05
aps	0.80	0.79	0.78	0.77	3.14	0.79	1.04
aps-apd (PADC)	0.80	0.79	0.78	0.77	3.14	0.79	1.04

Table 9. Effect on four identical prefetch-friendly (*libquantum*) applications on the 4-core system

Table 10 shows the system performance of PADC when four instances of a prefetch-unfriendly application, *milc*, are run together on the 4-core system. Because prefetches generated for each instance are useless most of the execution time of *milc*, demand-first and APS outperform demand-pref-equal for each instance. Incorporating APD into APS (i.e. PADC) further improves individual speedup of all instances equally by reducing useless prefetches from each instance. As a result, PADC significantly improves all system performance metrics. In fact, using PADC allows the system to gain significant performance improvement from prefetching whereas using a rigid prefetch scheduling policy results in a large performance loss due to prefetching. To conclude, PADC is also very effective when multiple identical applications run together on a CMP system.

	Individual speedup				WS	HS	UF
	milc	milc	milc	milc			
no-pref	0.53	0.53	0.53	0.53	2.11	0.53	1.00
demand-first	0.52	0.51	0.50	0.46	1.99	0.50	1.13
demand-pref-equal	0.36	0.36	0.36	0.36	1.45	0.36	1.01
aps	0.52	0.51	0.50	0.46	1.99	0.50	1.14
aps-apd (PADC)	0.59	0.58	0.58	0.58	2.33	0.58	1.02

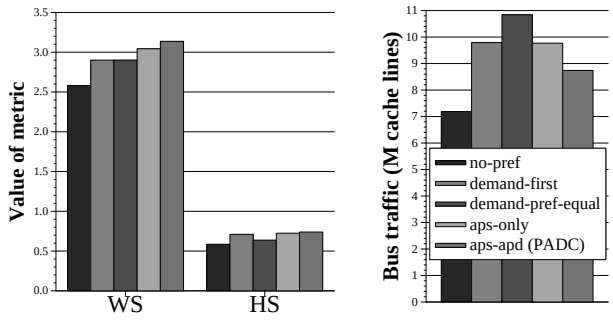
Table 10. Effect on four identical prefetch-unfriendly (*milc*) applications on the 4-core system

6.3.6. Overall Performance Figure 16 shows the average system performance and bus traffic for the 32 workloads run on the 4-core system. PADC provides the best performance and lowest bandwidth consumption compared to all previous prefetch handling policies. It improves weighted speedup and harmonic mean of speedups by 8.2% and 4.1%, respectively, compared to the demand-first policy and reduces bus traffic by 10.1% over the best-performing policy (demand-first).

We found that PADC outperforms both the demand-first and demand-prefetch-equal policies for all but one workload we examined. The worst performing workload is the combination of *vpr, gamess, dealII*, and *calculix*. PADC's WS degradation is only 1.2% compared to the demand-first policy. These applications are either insensitive to prefetching (class 0) or not memory intensive (*vpr*).

6.4. 8-Core Results

Figure 17 shows average performance and bus traffic over the 21 workloads we simulated on the 8-core system. Note that the rigid prefetch scheduling policies actually cause stream prefetching to degrade performance in the 8-core system. The demand-first policy reduces performance by 1.2% and the demand-prefetch-equal policy by 3.0% compared to no prefetching. DRAM bandwidth becomes a lot more valuable with the increased number of cores because the cores put more pressure on the memory system. At any

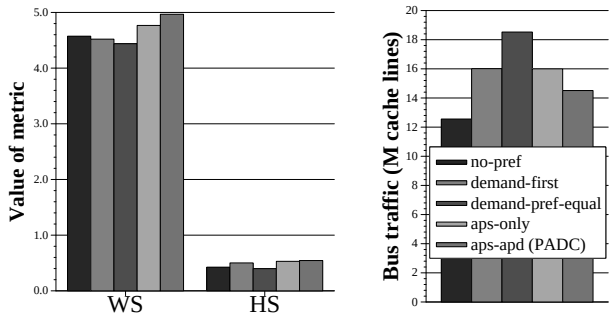


(a) Average system performance (b) Average traffic

Figure 16. Overall performance for 32 workloads on the 4-core system

given time there is a much larger number of demand and useful/useless prefetch requests in the memory request buffer. As a result, it becomes more likely that 1) a useless prefetch delays a demand or useful prefetch (if demand-prefetch-equal policy is used), and 2) DRAM throughput degrades if a demand request causes significant reduction in the row-buffer locality of prefetch requests (if demand-first policy is used). Hence, the degradation in performance with a rigid scheduling policy.

For the very same reasons, PADC becomes more effective when the number of cores increases. As resource contention becomes higher, the performance benefit of intelligent prioritization and dropping of useless prefetch requests increases. Our PADC improves overall system performance (WS) by 9.9% on the 8-core system while also reducing memory bandwidth consumption by 9.4%. We conclude that the benefits of PADC will continue to increase as off-chip memory bandwidth becomes a bigger performance bottleneck in future systems with many cores.



(a) Average system performance (b) Average traffic

Figure 17. Overall performance for 21 workloads on the 8-core system

6.5. Optimizing PADC for Fairness Improvement in CMP Systems: Incorporating Request Ranking

PADC can be better tuned and optimized for the requirements of CMP systems. One major issue in designing memory controllers for CMP systems is the need to ensure fair access to memory by different cores [20]. So far, we have considered PADC only as a way to improve overall system performance. However, to be more effective in CMP systems, PADC can be be augmented with a mechanism that provides fairness to different cores' requests. To achieve this purpose, this section describes a new scheduling algorithm that incorporates a request ranking scheme similar to the one used in Parallelism-Aware Batch Scheduling (PAR-BS) [20] into our Adaptive Prefetch Scheduling (APS) mechanism.

Recall that APS prioritizes urgent requests (demand requests from cores whose prefetch accuracy is low) over others to mitigate performance degradation and unfairness for prefetch-unfriendly applications. However, APS follows the FCFS policy if all other priorities (i.e. criticality, row-hit, urgency) are the same. This FCFS rule can degrade fairness and system performance by prioritizing requests of memory intensive applications over those of memory non-intensive applications, as was shown in previous work [23,

19, 20]. This happens because delaying the requests of memory non-intensive applications results in a lower individual speedup (or a higher slowdown) for those applications than it would for memory intensive applications which already suffer from long DRAM service time. Therefore, PADC (APS) itself cannot completely solve the unfairness problem. This is especially true in cases where all of the applications behave the same in terms of prefetch friendliness (either all are prefetch-friendly or all are prefetch-unfriendly). In such cases, PADC will likely degenerate into the FCFS policy (since the criticality, row-hit, and urgency priorities would be equal), resulting in high unfairness and performance degradation. For example, in case study II discussed in Section 6.3.2, all the applications are prefetch-unfriendly. Therefore, PADC prioritizes demands over prefetches most of the time. PADC mitigates performance degradation by prioritizing demand requests and dropping useless prefetches. However, *art* is very memory intensive and continuously generates many demand requests. These demand requests significantly interfere with other applications' demand requests, resulting in high slowdowns for the other applications while *art* itself experiences very little slowdown, thereby creating unfairness in the system.

To take into account fairness in PADC, we incorporate the concept of ranking, as employed in [20]. Our ranking scheme is based on the *shortest job first* principle [29] which can better mitigate the unfairness problem and performance degradation caused by the FCFS rule, as explained in detail in [20]. For each application, the DRAM controller keeps track of the total number of critical (demand and useful prefetch) requests in the memory request buffer. Applications with fewer outstanding critical requests are given a higher rank. The insight is that if an application that has fewer critical requests is delayed, the impact of that delay on that application's slowdown is much higher than the impact of delaying an application with a large number of critical requests. In other words, it is more unfair to delay an application that has a small number of useful requests (i.e., a "shorter" application/job) than delaying an application that has a large number of useful requests (i.e., a "longer" application/job). To achieve this while still being prefetch-aware, the DRAM controller schedules memory requests based on the modified rule shown in Rule 2. A highly-ranked request is scheduled by the DRAM controller when all requests in the memory request buffer have the same priority for criticality, row-hit, and urgency.

Rule 2 Adaptive Prefetch Scheduling with Ranking

1. **Critical request (C)**: Critical requests are prioritized over all other requests.
2. **Row-hit request (RH)**: Row-hit requests are prioritized over row-conflict requests.
3. **Urgent request (U)**: Demand requests generated by cores with low prefetch accuracy are prioritized over other requests.
4. **Highest rank request (RANK)**: Critical requests from a higher-ranked core are prioritized over critical requests from a lower-ranked core. Critical requests from cores that have fewer outstanding critical requests are ranked higher.
5. **Oldest request (FCFS)**: Older requests are prioritized over younger requests.

To implement this ranking mechanism, the priority field for each memory request is augmented as shown in Figure 18. To keep track of the total number of critical requests in the memory request buffer, a counter per core is required.[11] When the estimated prefetch accuracy of a core is greater than $promotion\_threshold$, the total number of outstanding demand and prefetch requests (critical requests) for that core are counted. When the accuracy is less than the threshold, the counter stores only the number of outstanding demand requests. Cores are ranked according to the total number of critical requests they have in the memory request buffer: a core that has a higher number of critical requests is ranked lower. The RANK field of a request is the same as the rank value of the core determined in this manner. As such, the critical requests of a core with a lower value in its counter are prioritized.[12] This

[11] We assume without loss of generality that each core can execute only one application/thread. If each core can execute multiple applications, our mechanism can simply be extended to distinguish between those applications in ranking.

[12] In this study, we do not rank non-critical requests (i.e. prefetch requests from cores whose prefetch accuracy is low). The RANK field of these requests is automatically set to 0, the lowest rank value. We evaluated a mechanism that also ranks non-critical requests based on estimated prefetch accuracy and found that this mechanism does not perform better than the mechanism that ranks only critical requests.

process is done every DRAM bus cycle in our implementation. Alternatively, determination of the ranking can be done periodically since it does not need to be highly accurate and is not on the critical path.

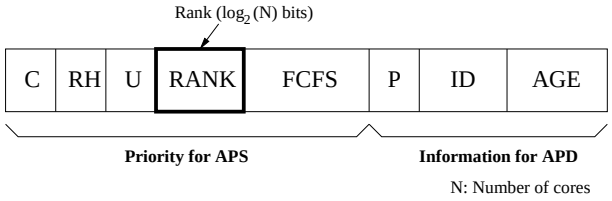
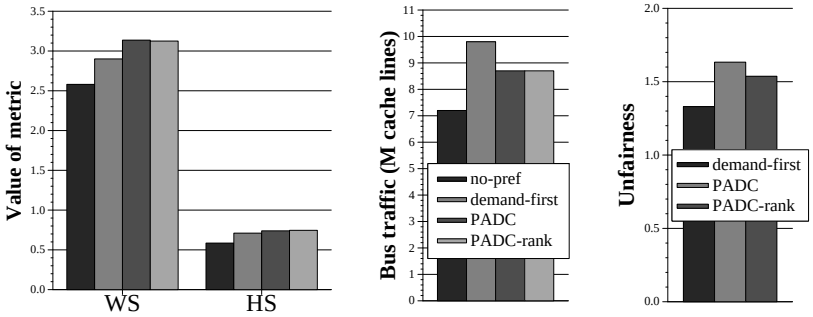


Figure 18. Memory request fields for PADC with ranking

Figure 19 shows the average system performance, bus traffic, and unfairness when we incorporate the ranking mechanism into PADC for the 32 4-core workloads. On average, the ranking mechanism slightly degrades weighted speedup (by 0.4%) and slightly improves harmonic mean of speedups (by 0.9%) and keeps bandwidth consumption about the same compared to the original PADC. Unfairness is improved from 1.63 to 1.53. The performance improvement is not significant because the contention in the memory system is not very high in the 4-core system. Nonetheless, the ranking scheme improves all the system performance and unfairness metrics for some workloads with memory intensive benchmarks. For the workload in case study II, the ranking scheme improves WS, HS, and UF by 7.5%, 10.3%, and 15.1% compared to PADC without ranking.



(a) Performance (b) Average traffic (c) Unfairness

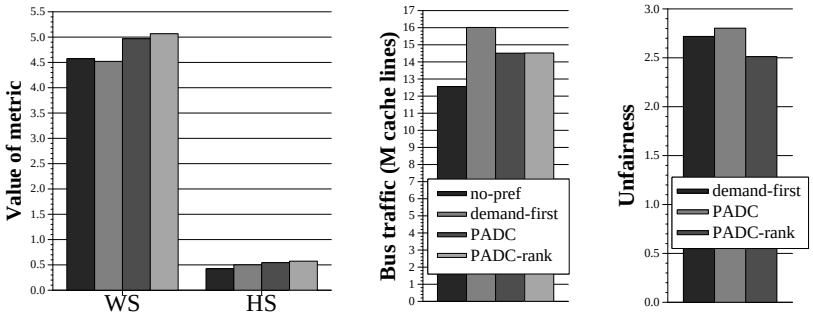
Figure 19. Optimized PADC using ranking mechanism on the 4-core system

We also evaluate the optimized PADC scheme with ranking on the 8-core system, which places significantly more pressure on the DRAM system. As shown in Figure 20, the ranking mechanism improves WS and HS by 2.0% and 5.4% respectively and reduces unfairness by 10.4% compared to PADC without ranking. The effectiveness of the ranking scheme is much higher in the 8-core system than the 4-core system since it is more critical to schedule memory requests fairly in many-core bandwidth-limited systems. Improving fairness reduces starvation of some cores, resulting in improved utilization of the cores in the system, which in turn results in improved system performance. Since starvation is more likely when the memory system is shared between 8 cores rather than 4, the performance improvement obtained with the ranking scheme is higher in the 8-core system.

We conclude that augmenting PADC with an intelligent fairness mechanism improves both unfairness and system performance.

6.6. Effect on Multiple DRAM Controllers

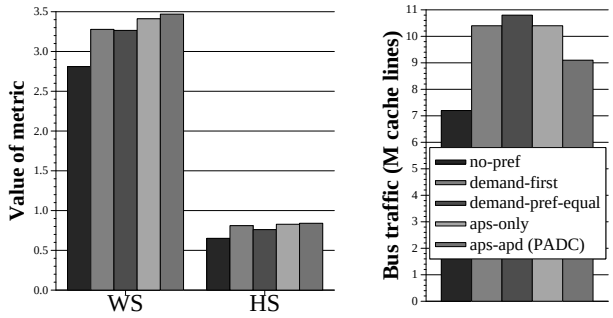
We also evaluate the performance impact of PADC when two DRAM controllers are employed in the 4- and 8-core systems. Each memory controller works independently through a dedicated channel (address and data buses) doubling the peak memory bandwidth. Because there is more bandwidth available in the system, contention of prefetch and demand requests is also significantly reduced. Therefore, the baseline system performance is significantly improved compared to the single controller. Adding one more DRAM controller improves weighted speedup by 16.9% and 30.9% compared to the single controller for 4- and 8-core systems, respectively.



(a) Performance (b) Average traffic (c) Unfairness

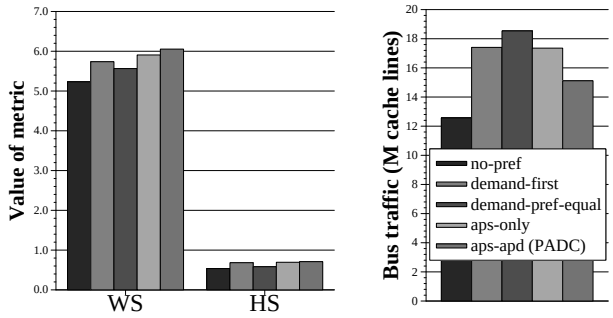
Figure 20. Optimized PADC using ranking mechanism on the 8-core system

Figures 21 and 22 show the average performance and bus traffic for 4- and 8-core systems with two memory controllers. Note that for the 8-core system, unlike the single memory controller configuration (shown in Figure 17(a)) where adding a prefetcher actually degrades performance, performance increases when adding a prefetcher even for the rigid scheduling policies because of the increased memory bandwidth.



(a) Average system performance (b) Average traffic

Figure 21. Overall performance for 32 workloads on the 4-core system with dual memory controllers



(a) Average system performance (b) Average traffic

Figure 22. Overall performance for 21 workloads on the 8-core system with dual memory controllers

PADC is still very effective with two memory controllers and improves weighted speedup by 5.9% and 5.5% and also reduces bandwidth consumption by 12.9% and 13.2% compared to the demand-first policy for 4- and 8-core systems respectively. Therefore, we conclude that PADC still performs effectively on a multi-core processor with very high DRAM bandwidth.

6.7. Effect with Different DRAM Row Buffer Sizes

As motivated in Section 3, PADC takes advantage of and relies on the row buffer locality of demand and prefetch requests generated at runtime. To determine the sensitivity of PADC to row buffer size, we varied the size of the row buffer from 2KB to

128KB for the 32 workloads run on the 4-core system. Figure 23 shows the WS improvements of PADC and APS compared to no prefetching, demand-first, and demand-prefetch-equal.

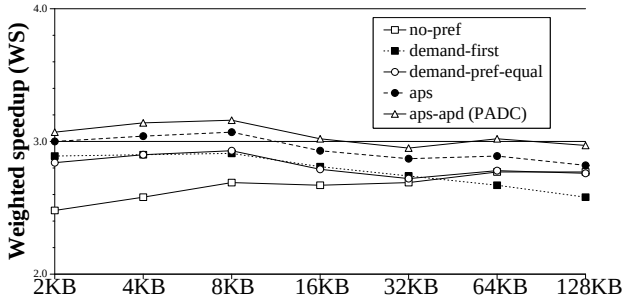


Figure 23. Effect of PADC with various DRAM row buffer sizes on the 4-core system

PADC consistently outperforms no prefetching, demand-first, and demand-prefetch-equal with various row buffer sizes. Note that the demand-first policy starts degrading performance compared to no prefetching as the row buffer becomes very large (more than 64KB). This is because preserving row buffer locality for useful requests is more critical when the row buffer size is large, especially when the stream prefetcher is enabled. No prefetching with larger row buffer sizes exploits row buffer locality more (higher row-hit rate) than smaller row buffer sizes. However, with demand-first, the negative performance impact of frequent re-activations of DRAM rows for demand and prefetch requests becomes significantly worse at larger row buffer sizes. Therefore, the demand-first policy experiences a higher memory service time on average than no prefetching with large row buffer sizes.

Similarly, the demand-prefetch-equal policy does not improve performance compared to no prefetching for 64KB and 128KB row buffer sizes since it does not take into account the usefulness of prefetches. With a large row buffer, useless prefetches have higher row buffer locality because many of them hit in the row buffer due to the streaming nature of the prefetcher. As a result, demand-prefetch-equal significantly delays the service of demand requests at large row buffer sizes by servicing more useless row-hit prefetches first.

In contrast to these two rigid scheduling policies, PADC tries to service only useful row-hit memory requests first, thereby significantly improving performance even for large row buffer sizes (8.8% and 7.3% compared to no prefetching for 64KB and 128KB row buffers). Therefore, PADC can make a prefetcher viable and effective even when a large row buffer size is used because it takes advantage of the increased row buffer locality opportunity provided by a larger row buffer *only* for useful requests instead of wasting the increased amount of bandwidth enabled by a larger row buffer on useless prefetch requests.

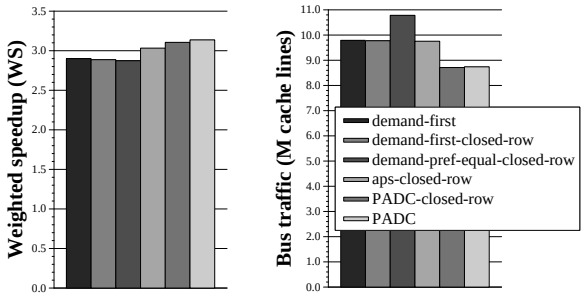
6.8. Effect with a Closed-Row DRAM Row Buffer Policy

So far we have assumed that the DRAM controller employs the open-row policy (i.e., it keeps the accessed row open in the row buffer after the access, even if there are no more outstanding requests requiring the row). In this section, we evaluate the effectiveness of PADC with a closed-row policy. The closed-row policy closes (by issuing a precharge command) the currently-opened row when all row-hit requests in the memory request buffer have been serviced by the DRAM controller. This policy can hide effective precharge time by 1) overlapping the precharge latency with the row-access latency [15] and 2) issuing the precharge command (closing a row buffer) earlier than the open-row policy (and thereby converting a row-conflict access into a row-closed access, if the next access is to a different row). Therefore, if no more requests to the same row arrive at the memory request buffer after a row buffer is closed by a precharge command, the closed-row policy can outperform the open-row policy. This is because, with the closed-row policy, the later requests do not need a precharge before activating the different row. However, if a request to the same row arrives at the memory request buffer soon after the row is closed, this policy has to pay a penalty (the sum of

the non-overlapped precharge latency and the activation latency), which would not have been required for the open-row policy. Consequently, for applications that have high row buffer locality (i.e. applications that generate bursty row-hit requests), such as streaming/striding applications, the open-row policy outperforms the closed-row policy by reducing re-activations of the same rows that will be needed again in the near future.

Since the closed-row policy still services row-hit requests first until no more requests to the same row remain in the memory request buffer, it can increase DRAM throughput within the scope of the requests that are outstanding in the memory request buffer. When a prefetcher is enabled with the closed-row policy, the same problem exists as for the open-row policy: none of the rigid prefetch scheduling policies can achieve the best performance for all applications since they are not aware of prefetch usefulness. Therefore PADC can still work effectively with the closed-row policy, as we show empirically below.

Figure 24 shows the performance and bus traffic when PADC is used with the closed-row policy for the 32 4-core workloads. The closed-row policy with demand-first scheduling slightly degrades performance by 0.5% compared to the open-row policy with demand-first scheduling. This is because there is a large number of streaming/striding (and prefetch-friendly) applications in the SPEC 2000/2006 benchmarks whose performance can be significantly improved with the open-row policy. The performance improvement of the open-row policy is not very significant because there is also a large number of applications that work well with the closed-row policy as they do not have high row buffer locality.



(a) Average system Performance

(b) Average traffic

Figure 24. Effect on closed-row policy

The results show that PADC is still effective with the closed-row policy since it still effectively exploits row buffer locality (within the scope of the requests outstanding in the memory request buffer) and reduces the negative effects of useless prefetch requests. PADC improves weighted speedup by 7.6% and reduces bandwidth consumption by 10.9% compared to demand-first scheduling with the closed-row policy. Note that PADC with the open-row policy slightly outperforms PADC with the closed-row by 1.1% for weighted speedup. Overall, we conclude that PADC is suitable for different row buffer management policies, but it is more effective with the open-row policy due to the existence of a larger number of benchmarks with high row buffer locality.

6.9. Effect with Different Last-Level Cache Sizes

PADC aims to maximize DRAM throughput for useful memory (demand and useful prefetch) requests and to delay and drop useless memory requests. One might think that a prefetch/demand management technique such as PADC would not be needed for larger L2 (last-level) caches since a larger cache can reduce cache misses (i.e. memory requests). However, a prefetcher can still generate a significant amount of useful prefetch requests for some applications or program phases by correctly predicting demand access patterns which cannot be stored even in large caches due to the large working set size or streaming nature of the program. In addition, the prefetcher can issue a significant number of useless prefetches for some other applications or program phases. For

these reasons, the interference between demands and prefetches still exists in systems with large caches. Therefore, we hypothesize PADC is likely to be effective in systems with large last-level caches.

To test this hypothesis, we evaluated the effectiveness of PADC for various L2 cache sizes. We varied the L2 cache size from 512KB to 8MB per core on our 4-core CMP system. Figure 25 shows the system performance (weighted speedup) for the 32 4-core workloads.

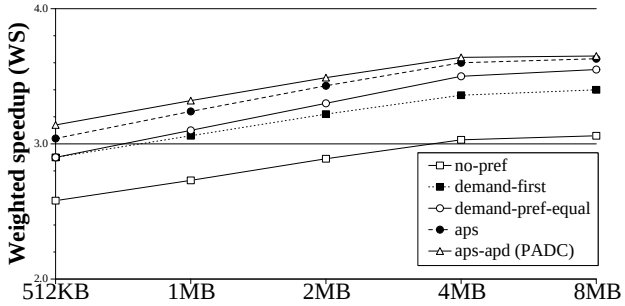


Figure 25. Effect of PADC on various cache sizes per core on the 4-core system

As expected, with larger cache sizes, baseline system performance improves. However, the stream prefetcher still effectively improves performance compared to no prefetching with either the demand-first and the demand-prefetch-equal policy. In addition, PADC consistently and significantly improves performance compared to both demand-first and demand-prefetch-equal policies regardless of cache size. This is mainly because even with large caches there is still a significant number of both useful and useless prefetches generated. Therefore, the interference between prefetch and demand requests still needs to be intelligently controlled.

There are two notable observations from Figure 25: 1) the demand-pref-equal policy starts outperforming the demand-first policy for caches greater than 1MB, and 2) the performance of APS (without APD) becomes closer to that of PADC (APS and APD together) as the cache size becomes larger.

Both observations can be explained by two reasons. First, a larger cache reduces irregular (or hard-to-prefetch) conflict cache misses due to the increased cache capacity. This makes the prefetcher more accurate because it reduces the allocations of stream entries for hard-to-prefetch access patterns (note that only a demand cache miss allocates a stream prefetch entry as discussed in Section 2.3). Second, a larger cache can tolerate some degree of cache pollution. Due to the increased cache capacity, the probability of replacing a demand or useful prefetch line with a useless prefetch in the cache is reduced.

For these reasons, the effect of deprioritizing or dropping likely-useless prefetches becomes less significant with a larger cache. As a result, as cache size increases, techniques that prioritize demands (e.g. demand-first) and drop prefetches (APD) start becoming less effective. However, the interference between prefetch and demand requests is not completely eliminated since some applications still suffer from useless prefetches. PADC (and APS) is effective in reducing this interference in systems with large caches and therefore still performs significantly better than the rigid scheduling policies.

Note that PADC with a 512KB L2 cache per core performs almost the same as demand-first with a 2MB L2 cache per core. Thus, PADC, which requires only 4.25KB storage, achieves the equivalent performance improvement that an additional 6MB (1.5MB $\times$ 4 (cores)) of cache storage would provide in the 4-core system.

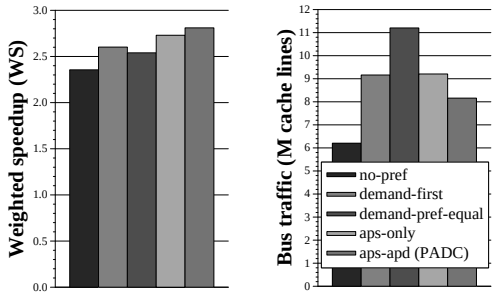
6.10. Effect with a Shared Last-Level Cache

Throughout the paper, we evaluate our mechanism on CMP systems with private on-chip last-level (L2) caches rather than a shared cache where all cores share a large on-chip last-level cache. This allowed us to easily show and analyze the effect of PADC in the shared DRAM system by isolating the effect of contention in the DRAM system from the effect of interference in shared caches.

However, many commercial processors already employ shared last-level caches in their CMP designs [34, 35]. In this section, we evaluate the performance of PADC in on-chip shared L2 caches on the 4- and 8-core systems to show the effectiveness of PADC in systems with a shared last-level cache.

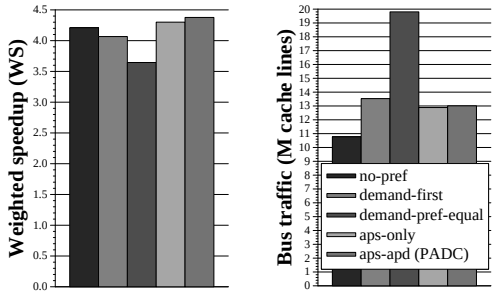
For this experiment, we use a shared L2 cache whose size is equivalent to the sum of all the private L2 cache sizes in our baseline system. We scaled the associativity of the shared cache with the number of cores on the chip since as the number of cores increases, the contention for a cache set increases. Therefore the 4-core system employs a 2MB, 16 way set-associative cache and the 8-core system has a 4MB, 32 way set-associative cache.

Figures 26 and 27 show weighted speedup and average bus traffic on the 4- and 8-core systems with shared L2 caches. PADC outperforms demand-first by 8.0% and 7.6% on the 4- and 8-core systems respectively. We conclude that PADC works efficiently for shared last-level caches as well.



(a) Performance (b) Average traffic

Figure 26. Effect on shared L2 cache on 4-core system



(a) Performance (b) Average traffic

Figure 27. Effect on shared L2 cache on 8-core system

Note that the demand-prefetch-equal policy does not work well on either of the shared cache systems (degrading WS by 2.4% and 10.4% compared to demand-first for 4- and 8-core systems). This is because the contention in the shared cache among the requests from different cores significantly increases compared to that of a private cache system. With private caches, useless prefetches from one core can only replace useful lines of that same core. However, with a shared cache, useless prefetches from one core can also replace the useful lines of all the other cores. These replaced lines must be brought back into the cache again from DRAM when they are needed. Therefore, the total bandwidth consumption significantly increases. This cache contention among cores becomes especially worse with demand-prefetch-equal for prefetch-unfriendly applications. This is because the demand-prefetch-equal policy results in high cache pollution since it blindly prefers to increase DRAM throughput without considering the usefulness of prefetches. The demand-prefetch-equal policy increases bus traffic by 22.3% and 46.3% compared to demand-first for the 4- and 8-core systems as shown in Figures 26(b) and 27(b). In contrast, PADC delays the service of useless prefetches and also drops them,

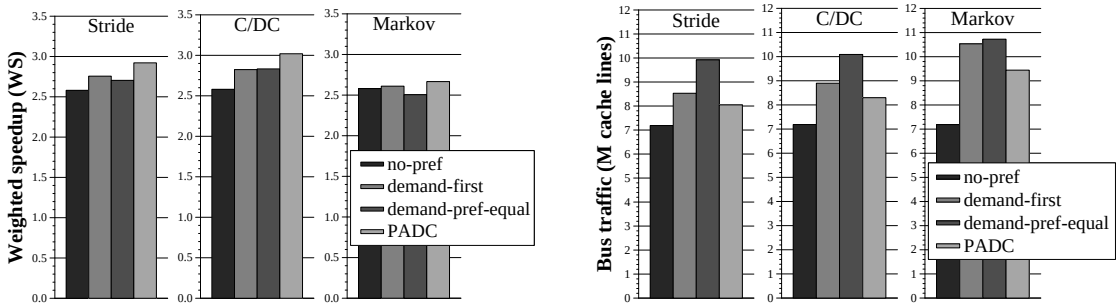
thereby mitigating contention in both the shared cache and the shared DRAM system.

6.11. Effect on Other Prefetching Mechanisms

We briefly evaluate the effect of our PADC on different types of prefetchers: PC-based stride [1], CZone Delta Correlation (C/DC) [24], and the Markov prefetcher [7]. Figure 28 shows the performance and bus traffic results averaged over all 32 workloads run on the 4-core system with the three different prefetchers. PADC consistently improves performance and reduces bandwidth consumption compared to the demand-first or demand-prefetch-equal policies with all three prefetchers.

The PC-based stride and C/DC prefetchers successfully capture a significant amount of memory access patterns as the stream prefetchers does, thereby increasing the potential for exploiting row buffer locality. In addition, those prefetchers also generate many useless prefetches for certain applications. Therefore, PADC significantly improves performance and bandwidth efficiency with these prefetchers by increasing DRAM throughput for useful requests and reducing the negative impact of useless prefetches.

The performance improvement of PADC on the Markov prefetcher is the least. This is because the Markov prefetcher, which exploits temporal as opposed to spatial correlation, does not work as well as the other prefetchers for the SPEC benchmarks. It generates many useless prefetches, which lead to significant waste/interference in DRAM bandwidth, cache space, and memory queue resources. Furthermore, it does not generate many useful prefetches for the SPEC benchmarks, and therefore its maximum potential for performance improvement is low. As such, the Markov prefetcher significantly increases bandwidth consumption and results in little performance improvement compared to no prefetching as shown in Figure 28(b). PADC improves the performance of the Markov prefetcher (mainly due to APD) by removing a large number of useless prefetches while keeping the small number of useful prefetches. PADC improves WS by 2.2% and reduces bandwidth consumption by 10.3% (mainly due to APD) compared to the demand-first policy. We conclude that PADC is effective with a wide variety of prefetching mechanisms.



(a) Performance (b) Traffic

Figure 28. PADC on stride, C/DC, and Markov prefetchers

6.12. Comparison with Dynamic Data Prefetch Filtering and Feedback Directed Prefetching

Dynamic Data Prefetch Filtering (DDPF) [41] tries to eliminate useless prefetches based on whether or not the prefetch was useful in the past. It records either the past usefulness of the prefetched address (or the PC of the instruction which triggered the prefetch) in a table similar to how a two-level branch predictor stores history information. When a prefetch request is created, the history table is consulted and the previous usefulness information is used to determine whether or not to send out the prefetch request. Feedback Directed Prefetching (FDP) [32] adaptively adjusts the aggressiveness of the prefetcher in order to reduce its negative effects.

Recall that PADC has two components: APS (Adaptive Prefetch Scheduling) and APD (Adaptive Prefetch Dropping). Both DDPF and FDP are orthogonal to APS because they do not deal with the scheduling of prefetches with respect to demands. As such,

they can be employed together with APS to maximize the benefits of prefetching. On the other hand, the benefits of DDPF, FDP, and APD overlap. DDPF filters out useless prefetches before they are sent to the memory system. FDP eliminates useless prefetches by reducing the aggressiveness of the prefetcher thereby reducing the likelihood that useless prefetch requests are generated. In contrast, APD eliminates useless prefetches by dropping them *after* they are generated. As a result, we find (based on our experimental analyses) that APD has the following advantages over DDPF and FDP:

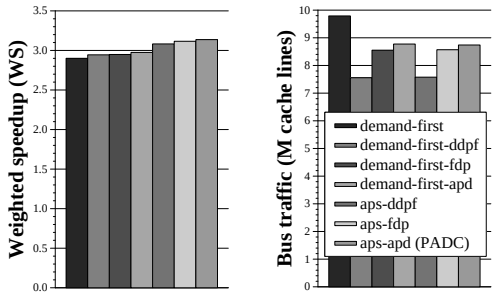
1. Both DDPF and FDP eliminate not only useless prefetches but also a significant fraction of useful prefetches. DDPF removes many useful prefetches by falsely predicting many useful prefetches to be useless. This is due to the aliasing problem caused by sharing the limited size of the history table among many addresses. FDP can eliminate useful prefetches when it reduces the aggressiveness of the prefetcher. In addition, we found that FDP can be very slow in increasing the aggressiveness of the prefetcher when a new phase starts execution. In such cases, FDP cannot issue useful prefetches whereas APD would have issued them because it always keeps the prefetcher aggressive.

2. The hardware cost of DDPF for an L2 cache is expensive since each L2 cache line and MSHR must carry several bits for indexing the prefetch history table (PHT) to update the table appropriately. For example, for a PC-based gshare DDPF with a 4K-entry PHT, 24 bits (12-bit branch history and 12-bit load PC bits) per cache line are needed in addition to the prefetch bit per cache line. For the 4-core system we use, this index information alone accounts for 96KB storage. In contrast, APD does not require significant hardware cost as we have shown in Section 4.4.

3. FDP requires the tuning of multiple threshold values [32] to throttle the aggressiveness of the prefetcher, which is a non-trivial optimization problem. APD allows the baseline prefetcher to *always* be very aggressive because it can eliminate useless prefetches after they are generated. As such, there is no need to tune multiple different threshold values in APD because the aggressiveness of the prefetcher never changes.

To evaluate the performance of these mechanisms, we implemented DDPF (PC-based gshare DDPF for L2 cache prefetch filtering [41]) and FDP in our CMP system. All the relevant parameters (FDP: prefetch accuracy (90%, 40%), lateness (1%), and pollution (0.5%) thresholds and pollution filter size (4Kbits); DDPF: filtering threshold (3), table size (4K entry 2-bit counters), and other train/predict policies) for DDPF and FDP were tuned for the best performance with the stream prefetcher in our CMP system. Figure 29 shows the performance and bus traffic of different combinations of DDPF, FDP, and PADC averaged across the 32 workloads run on the 4-core system. From left to right, the seven bars show: 1) baseline stream prefetching with the rigid demand-first policy, 2) DDPF with demand-first policy, 3) FDP with demand-first policy, 4) APD with demand-first policy, 5) DDPF combined with APS, 6) FDP combined with APS, and 7) APD combined with APS (i.e. PADC). When used with the demand-first policy, DDPF and FDP improve performance by 1.5% and 1.7% respectively while reducing bus traffic by 22.8% and 12.6%. In contrast, APD improves performance by 2.6% while reducing bus traffic by 10.4%. DDPF and FDP eliminate more useless prefetches than APD resulting in less bus traffic. However, for the very same reason, DDPF and FDP eliminate many useful prefetches as well. Therefore, their performance improvement is not as high as APD.

Our adaptive scheduling policy and DDPF/FDP are complementary and improve performance significantly when combined together. When used together with Adaptive Prefetch Scheduling, DDPF and FDP improve performance by 6.3% and 7.4% respectively. Finally, the results show that PADC outperforms the combination of DDPF/FDP and APS, which illustrates that Adaptive Prefetch Dropping is better suited to eliminate the negative performance effects of prefetching than DDPF and FDP. We conclude that 1) our adaptive scheduling technique complements DDPF and FDP whereas our APD technique outperforms DDPF and FDP, and 2) DDPF and FDP reduce bandwidth consumption more than APD, but they do so at the expense of performance.

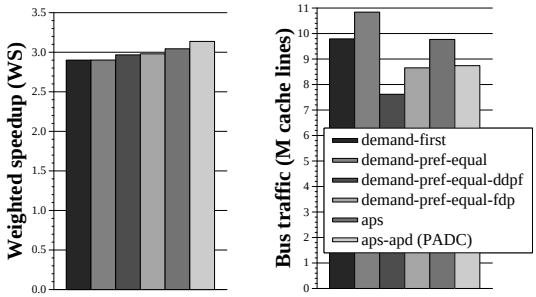


(a) Performance (b) Average traffic

Figure 29. Comparison with DDPF and FDP with demand-first scheduling

If a prefetch filtering mechanism is able to eliminate all useless prefetches while keeping all useful prefetches, the demand-prefetch-equal policy would be best performing. That is, we do not need an adaptive memory scheduling policy since all prefetches sent to the memory system would be useful. However, it is not trivial to design such perfect prefetch filtering mechanism. As discussed above, DDPF and FDP filter out not only useless prefetches but also a lot of useful prefetches. Therefore, combining those schemes with demand-prefetch-equal does not necessarily significantly improve performance since benefits of useful prefetches are reduced.

Figure 30 shows performance and average traffic when DDPF and FDP are combined with demand-prefetch-equal. Since DDPF and FDP remove a significant amount of useful prefetches, the performance improvement of them is not very significant (only by 2.3% and 2.7% compared to demand-first). On the other hand, PADC significantly improve performance (by 8.2%) by keeping the benefits of useful prefetches as much as possible.



(a) Performance (b) Average traffic

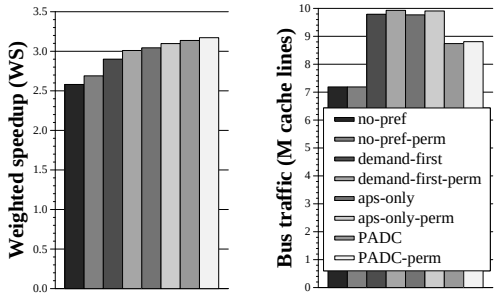
Figure 30. Comparison to DDPF and FDP with demand-prefetch-equal scheduling

6.13. Effect with a DRAM Bank Remapping Scheme

Permutation-based page interleaving [38] aims to reduce row-conflicts by randomly remapping the DRAM bank indexes of addresses so that they are more spread out across the multiple banks in the memory system. This technique significantly improves DRAM throughput by increasing utilization of multiple DRAM banks. The increased utilization of the banks has the potential to reduce the interference between memory requests. However, this technique cannot completely eliminate the interference between demand and prefetch requests in the presence of prefetching. Any rigid prefetch scheduling policy in conjunction with this technique will still have the same problem we describe in Section 3: none of the rigid prefetch scheduling policies can achieve the best performance for all applications since they are not aware of prefetch usefulness. Therefore, PADC is complementary to permutation-based page interleaving.

Figure 31 shows the performance impact of PADC for the 32 4-core workloads when a permutation-based interleaving scheme is applied. The Permutation-based scheme improves system performance by 3.8% over our baseline with the demand-first policy.

APS and PADC consistently work effectively combined with the permutation-based interleaving scheme. APS and PADC improve system performance by 2.9% and 5.4% repectively compared to the demand-first policy with the permutation-base interleaving scheme. Also, PADC reduces bandwidth consumption by 11.3% due to adaptive prefetch dropping.



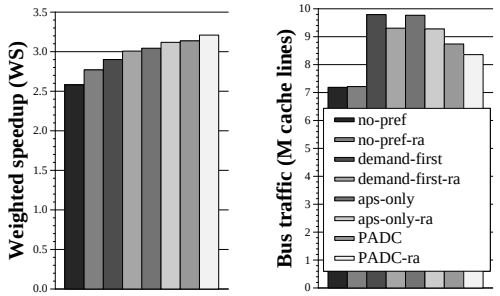
(a) Performance (b) Average traffic

Figure 31. Effect on permutation-based page interleaving

6.14. Effect on a Runahead Execution Processor

Runahead execution is a promising technique that prefetches useful data by executing future instructions that are independent of a long latency (runahead-causing) load instruction during the stall time of the load instruction. Because it is based on the execution of actual instructions, runahead execution can prefetch irregular data access patterns as well as regular ones. Usually, runahead execution complements hardware prefetching and results in high performance. In this section, we analyze the effect of PADC on a runahead processor. We implemented runahead capability in our CMP simulator. Since memory requests during runahead modes are very accurate most of the time [21], we treat runahead requests the same as demand requests in DRAM scheduling.

Figure 32 shows the effect of PADC on a runahead processor for the 32 workloads on the 4-core CMP system. Adding runahead execution on top of the baseline demand-first policy improves system performance by 3.7% and also reduces bandwidth consumption by 5.0%. This is because we use a prefetcher update policy that trains existing stream prefetch entries but does not allocate a new stream prefetch entry on a cache miss during runahead execution (*only-train*). Previous research [18] shows that this policy is best performing and most efficient. Runahead execution with the only-train policy can make prefetching more accurate and efficient by capturing irregular cache misses during runahead execution. Furthermore, these irregular misses train existing stream prefetch entries, but new, more speculative, stream prefetch entries will not be created during runahead mode. This not only prevents the prefetcher from generating useless prefetches due to falsely created streams but also improves the accuracy and timeliness of the stream prefetcher since existing streams continue to be trained during runahead mode.



(a) Performance (b) Average traffic

Figure 32. Effect on runahead execution

Figure 32 shows that PADC still effectively improves performance by 6.7% and reduces bandwidth consumption by 10.2%

compared to a runahead CMP processor with the stream prefetcher and the demand-first policy. We conclude that PADC is effective at improving performance and bandwidth-efficiency for an aggressive runahead CMP by successfully reducing the interference between demand/runahead and prefetch requests in the DRAM controller.

7. Related Work

The main contribution of our work beyond previous research is an adaptive way of handling prefetch requests in the memory controller's scheduling and buffer management policies. To our knowledge, none of the previously proposed DRAM controllers adaptively prioritize between prefetch and demand requests nor do they adaptively drop useless prefetch requests based on prefetch usefulness information obtained from the prefetcher. We discuss closely related work in DRAM scheduling, prefetch filtering, and adaptive prefetching.

7.1. Prefetch Handling in DRAM Controllers

Many previous DRAM scheduling policies were proposed to improve DRAM throughput in single-threaded [42, 27, 5], multi-threaded [26, 22, 39], and stream-based [14, 37] systems. In addition, several recent works [23, 19, 20] proposed techniques for fair DRAM scheduling across different applications sharing the DRAM system. Some of these previous proposals [42, 27, 22, 39, 23, 19, 20] do not discuss how prefetch requests are handled with respect to demand requests. Therefore, our mechanism is orthogonal to these scheduling policies. These policies can be extended to adaptively prioritize between demand and prefetch requests and to adaptively drop useless prefetch requests.

The remaining DRAM controller proposals take two different approaches to handling prefetch requests. First, some proposals [11, 5, 6] always prioritize demand requests over prefetch requests. Other proposals [37, 26] treat prefetch requests the same as demand requests. As such, these previous DRAM controller proposals handle prefetch requests rigidly. As we have shown in Sections 1 and 3, rigid handling of prefetches can cause significant performance loss compared to adaptive prefetch handling. Our work improves upon these proposals by incorporating the effectiveness of prefetching into DRAM scheduling decisions.

7.2. Prefetch Filtering

Our Adaptive Prefetch Dropping (APD) scheme shares the same goal of eliminating useless prefetches with several other previous proposals. However, our mechanism provides either higher bandwidth-efficiency or better adaptivity compared to these works.

Charney and Puzak [2] and Mutlu et al. [17] proposed prefetch filtering mechanisms using on-chip caches (using the tag store). Both of these proposals unnecessarily consume memory bandwidth since useless prefetches are filtered out only *after they are serviced by the DRAM system*. In contrast, APD eliminates useless prefetches before they consume valuable DRAM bandwidth.

Mowry et al. [16] proposed a mechanism that cancels software prefetches when the prefetch issue queue is full. Their mechanism is not aware of the usefulness of prefetches. On the other hand, our scheme drops prefetch requests only if their age is greater than a dynamically adjusted threshold (based on prefetch accuracy). Also, our mechanism can be applied to software prefetching to remove useless software prefetches more efficiently. Srinivasan et al. [33] use a profiling technique to mark load instructions that are likely to generate useful prefetches. This mechanism needs ISA support to mark the selected load instructions and cannot adapt to phase behavior in prefetcher accuracy. In contrast, APD does not require ISA changes and can adapt to changes in prefetcher accuracy.

Zhuang and Lee [40, 41] propose a mechanism that eliminates the prefetch request for an address if the prefetch request for the same address was useless in the past. PADC outperforms and also complements their mechanism as discussed in Section 6.12.

7.3. Adaptive Prefetching

Several previous works proposed changing the aggressiveness of the hardware prefetcher based on dynamic information. Our work is either complementary to or higher-performing than these proposals, as described below.

Hur and Lin [6] designed a probabilistic prefetching technique which adjusts prefetcher aggressiveness. They also schedule prefetch requests to DRAM adaptively based on the frequency of demand request DRAM bank conflicts caused by prefetch requests. However, their scheme always prioritizes demand requests over prefetches. In contrast, our mechanism adapts the prioritization policy between demands and prefetches based on prefetcher accuracy. As a result, Hur and Lin's proposal can be combined with our adaptive prefetch scheduling policy to provide even higher performance.

Srinath et al. [32] show how adjusting the aggressiveness of the prefetcher based on accuracy, lateness, and cache pollution information can reduce bus traffic without compromising the benefit of prefetching. As we showed in Section 6.12, PADC outperforms and also complements their mechanism.

8. Conclusion

This paper shows that existing DRAM controllers that employ rigid, non-adaptive prefetch scheduling and buffer management policies cannot achieve the best performance since they do not take into account the usefulness of prefetch requests. To overcome this limitation, we propose a low-cost Prefetch-Aware DRAM Controller (PADC), which aims to 1) maximize the benefit of useful prefetches by adaptively prioritizing them, and 2) minimize the harm caused by useless prefetches by adaptively deprioritizing them and dropping them from the memory request buffers. To this end, PADC dynamically adapts its memory scheduling and buffer management policies based on prefetcher accuracy. Our evaluation shows that PADC significantly improves system performance and bandwidth-efficiency on both single-core and multi-core systems. We conclude that incorporating awareness of prefetch usefulness into memory controllers is critical to efficiently utilizing valuable memory system resources in current and future systems.

References

[1] J. Baer and T. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, pages 178–186, 1991.
[2] M. Charney and T. Puzak. Prefetching and memory system behavior of the SPEC95 benchmark suite. *IBM Journal of Research and Development*, 31(3):265–286, 1997.
[3] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, 2008.
[4] J. D. Gindele. Buffer block prefetching method. *IBM Technical Disclosure Bulletin*, 20(2):696–697, July 1977.
[5] I. Hur and C. Lin. Adaptive history-based memory scheduler. In *MICRO-37*, 2004.
[6] I. Hur and C. Lin. Memory prefetching using adaptive stream detection. In *MICRO-39*, 2006.
[7] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *ISCA-24*, pages 252–263, 1997.
[8] T. Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *Second Workshop on Memory Performance Issues*, 2002.
[9] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM power6 microarchitecture. *IBM Journal of Research and Development*, 51:639–662, 2007.
[10] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-aware DRAM controllers. In *MICRO-41*, 2008.
[11] W.-F. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *HPCA-7*, pages 301–312, 2001.
[12] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *ISPASS*, pages 164–171, 2001.
[13] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.
[14] S. A. McKee, W. A. Wulf, J. H. Aylor, R. H. Klenke, M. H. Salinas, S. I. Hong, and D. A. Weikle. Dynamic access ordering for streamed computations. *IEEE Computer*, 49:1255–1271, Nov. 2000.
[15] Micron. *2Gb DDR3 SDRAM, MT41J512M4 - 64 Meg x 4 x 8 banks*. http://download.micron.com/pdf/datasheets/dram/ddr3/.
[16] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, 1992.
[17] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt. Using the first-level caches as filters to reduce the pollution caused by speculative memory references. *International Journal of Parallel Programming*, 33(5):529–559, October 2005.
[18] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *ISCA-32*, 2005.
[19] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO-40*, 2007.
[20] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA-35*, 2008.
[21] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA-9*, 2003.
[22] C. Natarajan, B. Christenson, and F. Briggs. A study of performance impact of memory controller features in multi-processor server environment. In *WMPI*, pages 80–87, 2004.
[23] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO-39*, 2006.
[24] K. J. Nesbit, A. S. Dhodapkar, J. Laudon, and J. E. Smith. AC/DC: An adaptive data cache prefetcher. In *PACT-13*, 2004.
[25] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *MICRO-37*, 2004.
[26] S. Rixner. Memory controller optimizations for web servers. In *MICRO-37*, 2004.
[27] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA-27*, 2000.
[28] A. J. Smith. Cache memories. *Computing Surveys*, 14(4):473–530, 1982.

[29] W. E. Smith. Various optimizers for single stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.
[30] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. In *ASPLOS-9*, pages 164–171, 2000.
[31] L. Spracklen and S. G. Abraham. Chip multithreading: opportunities and challenges. In *HPCA-11*, pages 248–252, 2005.
[32] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA-13*, 2007.
[33] V. Srinivasan, G. S. Tyson, and E. S. Davidson. A static filter for reducing prefetch traffic. Technical Report CSE-TR-400-99, University of Michigan Technical Report, 1999.
[34] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Technical White Paper*, Oct. 2001.
[35] O. Wechsler. Inside Intel Core microarchitecture. *Intel Technical White Paper*, 2006.
[36] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *ISCA-19*, 1992.
[37] C. Zhang and S. A. McKee. Hardware-only stream prefetching and dynamic access ordering. In *ICS-14*, 2000.
[38] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *ISCA-27*, 2000.
[39] Z. Zhu and Z. Zhang. A performance comparison of DRAM memory system optimizations for SMT processors. In *HPCA-11*, 2005.
[40] X. Zhuang and H.-H. S. Lee. A hardware-based cache pollution filtering mechanism for aggressive prefetches. In *Proceedings of the 32nd Intl. Conference on Parallel Processing*, pages 286–293, 2003.
[41] X. Zhuang and H.-H. S. Lee. Reducing cache pollution via dynamic data prefetch filtering. *IEEE Transactions on Computers*, 56(1):18–31, Jan. 2007.
[42] W. Zuravleff and T. Robinbson. Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order. U.S. Patent Number 5,630,096, 1997.