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

§Microsoft Research and Carnegie Mellon University
onur@{microsoft.com,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 on both single- and multi-core systems with a variety of prefetching algorithms. 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*) [14]. Due to this non-uniform access latency, state-of-the-art memory access scheduling policies, such as [34, 24, 13], prefer row-hits over row-conflicts to improve DRAM throughput, thereby improving system performance. The problem of DRAM access scheduling becomes more challenging if we take prefetching 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. Existing DRAM scheduling policies take two different approaches as to how to treat prefetch requests with respect to demand requests. Some policies [31, 24] regard a prefetch request to have the same priority as a demand request. This can significantly delay demand requests and cause performance degradation, especially if prefetch requests are not accurate. Other policies [7, 11, 4, 27, 28] 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 doing so provides the best performance by eliminating the interference of prefetch requests with demand requests. However, such a rigid 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 higher DRAM throughput and can sometimes provide better system performance than 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 [30, 28] 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 [24] that prioritizes requests as follows: 1) row-hit requests over all others, 2) older requests over younger requests. 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 demand requests (or prefetch requests), the policy uses the same prioritization rules as the FR-FCFS policy. 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.

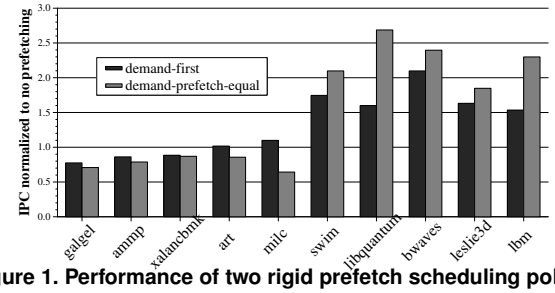


Figure 1. Performance of two rigid prefetch scheduling policies

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)

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

of prefetches, thereby improving system performance.[2] 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.[3]

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 (PADC)* on a wide variety of benchmarks and systems and find that it consistently outperforms previous DRAM controllers that rigidly handle prefetches on both single and multi-core systems. Our controller improves the performance of the 55 SPEC 2000/2006 benchmarks by 4.3% compared to the best previous controller on a single-core processor. Our mechanism also improves system performance (i.e., weighted speedup) for 32 SPEC workloads by 8.2% on a 4-core system, and for 21 workloads by 9.9% on an 8-core system while also reducing memory bandwidth consumption by 10.7% and 9.4% 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.

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.

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.

4. We show that the proposed adaptive scheduling and buffer-management mechanisms interact positively. Together, they significantly improve performance and bandwidth-efficiency on both single-core and multi-core systems with a variety of prefetching algorithms.

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. A DRAM access can be done only by reading (writing) data from (to) the row buffer using a column address.

There are three commands that need to be sequentially issued to a DRAM bank in order to access data: 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 [14].[4] Therefore, a memory controller can try to maximize DRAM data throughput by maximizing the hit rate in the row buffer. Previous work [24] introduced the commonly-employed FR-FCFS (First Ready-First Come First Serve) policy which 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 [24, 13]. However, 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 [30]. Stream prefetchers are commonly used in many processors [30, 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 appli-

[2] 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 explain this in more detail in Section 6.1.

[3] For completeness, we also implemented another policy, *prefetch-first*, that always prioritizes prefetch requests over demand requests. This policy provides the worst performance on all benchmarks: it degrades average IPC by 5.8% compared to the demand-first policy.

[4] Row-hit latency is about one third of the latency of a row-conflict for a contemporary SDRAM bank. For example, the row-hit and row-conflict latencies are 12.5ns and 37.5ns respectively for a 2Gbit DDR3 SDRAM chip [14]. The row-closed latency is 25ns. We use the open-row policy throughout the paper since it increases the possibility to improve DRAM throughput. The open-row policy provides 0.5% higher performance compared to the closed-row policy for our multiprogrammed 4-core workloads.

cation 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.[5] As such, stream prefetchers are likely to generate many row-hit prefetch requests which our prefetch-aware DRAM controller can take advantage of. We also evaluated our mechanism with three other prefetchers: a stride prefetcher [1], CZone/Delta Correlation (C/DC) prefetcher [22], and the Markov Prefetcher [7]. All of these prefetchers can generate a significant fraction of row-hit prefetch requests, especially for streaming/striding address/correlation patterns.

3. Motivation

None of the existing DRAM scheduling policies take into account both the non-uniform nature of DRAM access latencies and the usefulness of prefetch requests. Figure 2 illustrates why a rigid, non-adaptive prefetch scheduling policy degrades performance. 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.

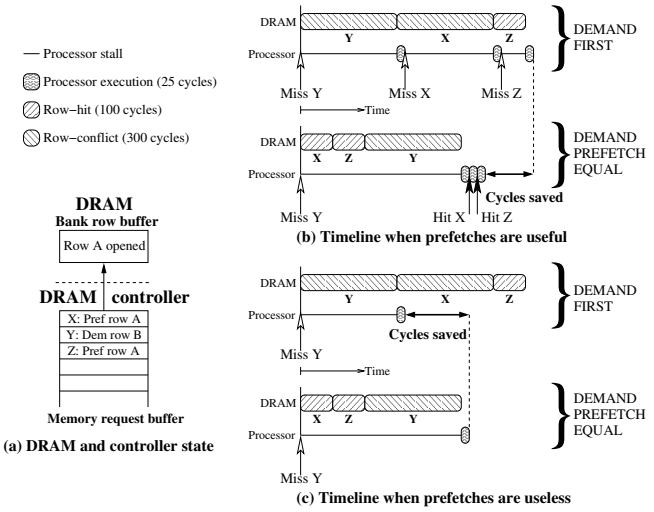


Figure 2. Example illustrating the performance impact of demand-first and demand-prefetch-equal policies

For Figure 2(b), assume that the processor needs to load addresses Y, X, and Z in a serial fashion (i.e., both of the prefetch requests 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.[6] We assume processor execution takes a small number of cycles because previous studies [19, 8] showed 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 demand-first (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). The processor then stalls for an additional row-hit latency since it requires the data 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.

With the demand-prefetch-equal policy (bottom), the row-hit prefetch requests to X and Z are satisfied first followed by the row-conflict demand request to Y. The processor must stall until the demand request to Y is serviced. However, after that, the processor only needs to perform the computations between the load instructions because loads to X and Z hit in the cache. The total execution time is the sum of one row-conflict latency and two row-hit latencies (plus a small period of processor execution), which is less than with the demand-first policy. 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 *swim, libquantum*, 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 resulting timeline. With demand-first, the processor stalls for only a single row-conflict latency which is required to service the demand request to Y. On the other hand, with demand-prefetch-equal, the processor stalls additional cycles since X and Z are serviced (even though they are not needed) before Y in the DRAM bank thereby delaying the useful request to Y. 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 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. 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 adaptively schedules prefetch and demand requests to increase DRAM throughput for useful requests. APD cancels useless prefetch requests 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. Therefore we first 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:

[5] The stream prefetcher we use for the evaluations is the best performing among the large set of prefetchers we examined. It improves performance by 20% on average for all the 55 SPEC 2000/2006 benchmarks using demand-first policy. For its detailed implementation, refer to [28, 30].

[6] 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.

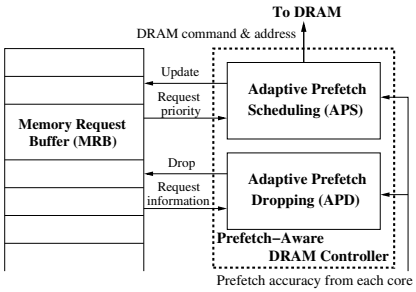


Figure 3. Prefetch-Aware DRAM Controller

1. Prefetch (P) bit per L2 cache line and memory request buffer (MRB) entry.[7] For a memory request buffer entry, 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 a cache line, this bit indicates whether or not the line was brought into the cache by a prefetch request. It is set when the line is filled (only if the P 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: keeps track of the total number of prefetch requests sent by the core. It is incremented when a prefetch request is sent to the memory request buffer by the core.

3. Prefetch Used Counter (PUC) per core: 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 or when a demand request matches a prefetch request in the memory request buffer.

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

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

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 requests and useful prefetch requests a higher priority than useless prefetch requests so that useless prefetch requests do not interfere with useful requests.

If the prefetch accuracy of a core is greater than or equal to a certain threshold, $promotion\_threshold$, all of the prefetch requests from that core are treated the same as demand requests. We call such prefetch requests and all demand requests *critical* requests. Otherwise, if the prefetch accuracy of a core is less than $promotion\_threshold$, then 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.

[7] Many previous proposals [3, 25, 33, 28] already use a prefetch bit for each cache line and memory request buffer entry.

Rule 1 Adaptive Prefetch Scheduling (APS)

1. **Critical request (C)**: Demand and useful prefetches are prioritized.
2. **Row-hit request (RH)**: Row-hits are prioritized over row-conflicts.
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 newer ones.

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 boost the demand requests of a core with low prefetch accuracy over the critical requests of cores with high prefetch accuracy. This is done for 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 prioritization rule). Doing so risks starving the demand requests of the core with low prefetch accuracy, resulting in a performance degradation since a large number of critical requests (demand *and* prefetch requests) from the core with high prefetch accuracy can contend with the critical requests (demand requests *only*) from the core with low prefetch accuracy. To avoid this, 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 useless prefetches. By prioritizing the demand requests of such 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 6.2.4.

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

4.3. Adaptive Prefetch Dropping

APS naturally delays 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. Although doing so reduces the interference of useless requests with useful requests, it cannot get rid of all of the negative effects of useless prefetch requests (e.g., bandwidth consumption and cache pollution) because such requests will eventually be serviced. 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, it is likely to be useless, and therefore, 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.

Why are old prefetch requests likely to be useless? Useful prefetches tend to have a shorter service time than useless prefetches. This is because a prefetch request that is waiting in the request buffer can become a demand request[8] if the processor sends a demand request for that same address while the prefetch request is still in the buffer. Such useful prefetches that become demand requests will be serviced earlier by both the demand-first prioritization policy and APS

[8] A prefetch request that is hit by a demand request in the memory request buffer becomes a 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.

(since APS treats all demand requests as critical). Therefore, useful prefetches on average experience a shorter service time than useless prefetches. For example, for *milc*, the average service time for useful prefetches is 1486 cycles compared to 2238 cycles for useless prefetches.

Mechanism: The observation that old prefetch requests are likely to be useless motivates us to remove a prefetch 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. 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). Note that APD interacts positively with APS since APS naturally delays the service of useless (non-critical) requests such that APD can remove them from the memory system.

Determining $drop\_threshold$: Figure 4 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 and therefore a dynamic $drop\_threshold$ value is needed. Between 150-275 million cycles, prefetch accuracy is very low (close to 0%), implying many useless prefetch requests were generated. 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 $drop\_threshold$ when prefetch accuracy is low to facilitate quick dropping of useless prefetch requests. On the other hand, we would want $drop\_threshold$ to be higher during periods of high prefetch accuracy to avoid prematurely dropping useful prefetch requests. Our evaluation shows that a simple 4-level $drop\_threshold$ that is dynamically adjusted can effectively reduce useless prefetch requests in the memory system while keeping useful prefetch requests.

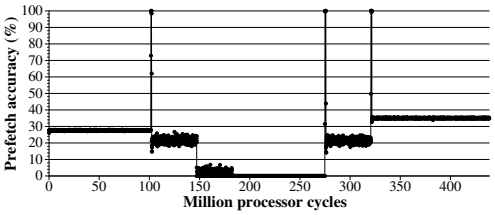


Figure 4. Phase behavior of prefetch accuracy in *milc*

4.4. Implementation and Hardware Cost of PADC

An implementation of PADC requires storing additional information in each memory request buffer entry to support the priority and aging information needed by APS and APD. Figure 5 shows the required additional information (in terms of the fields added to each request buffer entry). The C, RH, and FCFS fields are already used in the baseline demand-first FR-FCFS 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.

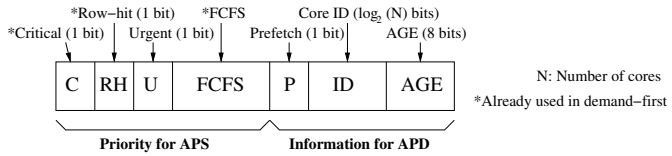


Figure 5. Fields of a memory request buffer entry in PADC

APD 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 and the request has not yet started being serviced in DRAM. Before removing a prefetch request, APD ensures that the prefetch cannot be matched by a demand request. This is accomplished by invalidating the MSHR entry of the prefetch request before actually dropping it. The APD unit uses the P, ID, and AGE fields of each request buffer entry. It determines the corresponding core's $drop\_threshold$ for each prefetch request and compares it to the AGE field of the request. 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 in our evaluation.

Table 1 shows the hardware storage cost required for our implementation of PADC. The total storage cost for our 4-core CMP system described in Section 5 is only 34,720 bits (∼4.25KB). 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	Cost equation (bits)	Cost (bits)
Prefetch accuracy	P (1 bit)	$N_{cache} \times N_{core} + N_{req}$	32,896
	PSC (16 bits)	$N_{core} \times 16$	64
	PUC (16 bits)	$N_{core} \times 16$	64
	PAR (8 bits)	$N_{core} \times 8$	32
APS	U (1 bit)	N_{req}	128
APD	ID ($log_2 N_{core}$ bits)	$N_{req} \times log_2 N_{core}$	256
	AGE (10 bits)	$N_{req} \times 10$	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 1. Hardware cost of PADC (N_{cache}: number of cache lines per core N_{core}: number of cores, N_{req}: number of request buffer entries)

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 at all levels of the memory hierarchy, as well as DDR3 DRAM system constraints. Table 2 shows the baseline configuration of each core. Table 3 shows the shared resource configuration for single-, 4-, and 8-core CMPs.

Execution core	Out of order; 15 stages; decode/retire up to 4 instructions, issue/execute up to 8 microinstructions 256-entry reorder buffer; 32-entry load-store queue
Front end	Fetch up to 2 branches; 4K-entry BTB; 64K-entry gshare, 64K-entry PAs, 64K-entry selector hybrid branch predictor
On-chip caches	L1 I and D: 32KB, 4-way, 2-cycle, 1 read and 1 write ports; Unified L2: 512KB (1MB for 1-core), 8-way, 8-bank, 15-cycle, 1 read/write port; 64B line size for all caches
Prefetcher	Stream prefetcher with 32 streams, prefetch degree of 4, cache line prefetch distance (lookahead) of 64 [30, 28]

Table 2. Baseline configuration of each core

DRAM controller	On-chip, demand-first FR-FCFS scheduling policy; 1 controller for 1-, 4-, 8-core CMP (also 2 for 8-core) 64, 128, 256-entry L2 MSHR and MRB for 1-, 4-, 8-core
DRAM and bus	DDR3 1333MHz [14], 16B-wide data bus per controller Latency: 15-15-15ns (^tRP,^tRCD, CL), BL = 4; 8 DRAM banks, 4KB row buffer per bank

Table 3. Baseline configuration of shared CMP resources

We use the SPEC CPU 2000/2006 benchmarks for experimental evaluation. Each benchmark was compiled using ICC (Intel C Compiler) or IFORT (Intel Fortran Compiler) with the -O3 option and was run with the reference input set for 200 million representative x86 instructions as selected by Pinpoints [23].

	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
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
ammp_00	1.70	0.80	1.47	1.70	56.20	5.96	8.03	2	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	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
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
milc_06	0.41	29.33	0.46	20.88	81.13	19.45	28.81	2	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

Table 4. Characteristics of some evaluated benchmarks RBH (Row Buffer Hit rate), MPKI (L2 Misses per 1000 inst.)

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.[9] The characteristics of a subset of benchmarks with and without a stream prefetcher are shown in Table 4. We chose only a subset of benchmarks due to limited space, but we do evaluate the entire set of 55 benchmarks in our single-core experiments. To evaluate our mechanism on CMP systems, we formed multiprogrammed combinations of the benchmarks. We ran 32 and 21 randomly chosen workload combinations (from the 55 SPEC benchmarks) for our 4- and 8-core CMP configurations respectively.

For the evaluation of PADC, we use a prefetch accuracy value of 85% for $promotion\_threshold$ (in APS) and the values shown in Table 5 for $drop\_threshold$ (in APD). Prefetch accuracy is calculated every 100K cycles.

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

Table 5. Dynamic $drop\_threshold$ values used in APD

5.2. Metrics

We use several metrics to measure bandwidth consumption and performance. *Bus traffic* is the total number of cache lines transferred over the bus during the execution of a workload. We define *prefetch accuracy (ACC)* and *coverage (COV)* as follows:

$$ACC = \frac{Num\ of\ useful\ prefetches}{Num\ of\ prefetches\ sent}$$

$$COV = \frac{Num\ of\ useful\ prefetches}{Num\ of\ demand\ requests + Num\ of\ useful\ prefetches}$$

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

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

To measure CMP system performance, we use *Individual Speedup (IS)*, *Weighted Speedup (WS)* [26], and *Harmonic mean of Speedups (HS)* [12]. 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 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 baseline demand-first policy to measure IPC^{alone} for all of our experiments.

$$IS_i = \frac{IPC_i^{together}}{IPC_i^{alone}}, \qquad WS = \sum_i^N IS_i, \qquad HS = \frac{N}{\sum_i^N \frac{1}{IS_i}}$$

[9] 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.

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 that 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-pref-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-pref-equal does better for most prefetch-friendly ones (class 1) such as *swim, libquantum*, and *lbm*. Averaged over all 55 benchmarks, the demand-pref-equal policy outperforms demand-first by 0.5% since there are more benchmarks (29 out of 55) that belong to class 1.

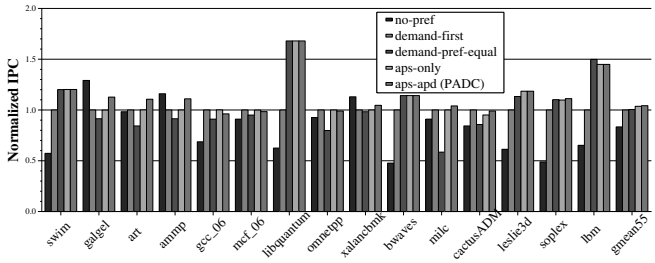


Figure 6. Single-core system performance (gmean55: all benchmarks)

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-pref-equal) improves performance over demand-first for many prefetch-friendly applications such as *libquantum, bwaves*, and *leslie3d*. This is due to two reasons. 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 requests get a chance to be issued. For example, APS improves prefetch coverage from 80%, 98%, and 89% to 100%, 100%, and 92% for *libquantum, bwaves*, and *leslie3d* respectively (as shown in Figure 8).

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 DRAM scheduling policy by itself cannot eliminate. Incorporating adaptive prefetch dropping (APD) into APS significantly improves performance especially in such 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). 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). PADC reduces SPL by 5.0% compared to the baseline by getting useless prefetches out of the way of useful requests. As a result, PADC significantly improves single-core performance.

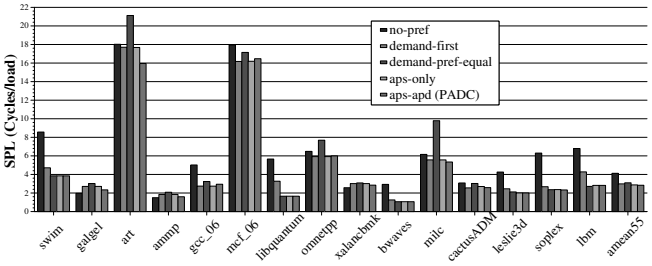


Figure 7. Stall time per load (SPL) on the single-core system

Figure 8 shows the bus traffic with each policy broken down into three categories: useful prefetches, useless prefetches, and demand requests. PADC reduces bus traffic by 10.4% on average across all benchmarks (amean55). Reduction in bus traffic is mainly due to APD, which significantly reduces the number of useless prefetches. For many benchmarks, APS by itself uses the same amount of bus bandwidth as the best rigid policy for each benchmark. We conclude that our prefetch-aware DRAM controller improves both performance and bandwidth-efficiency in single-core systems.

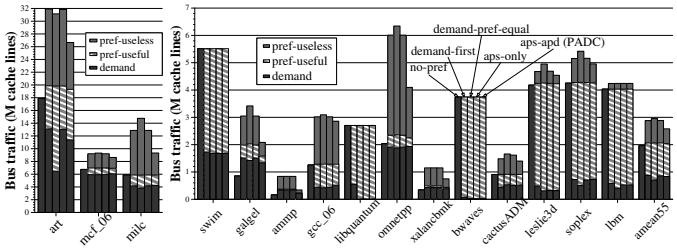
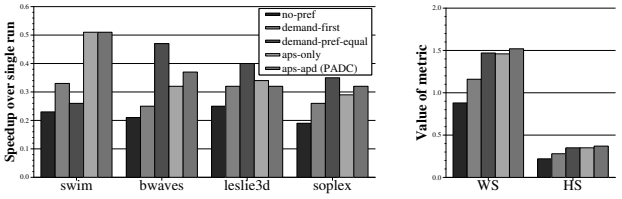


Figure 8. Bus traffic on the single-core system

6.2. 4-Core Results

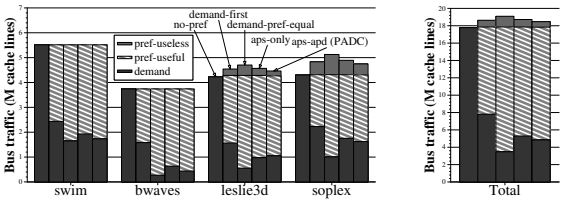
We ran 32 different workloads to evaluate the effectiveness of PADC on a 4-core system. In the following sections, we discuss three cases in detail to provide insights into the behavior of PADC.

6.2.1. Case Study I: 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. Figure 9 shows the speedup of each application and system performance. Figure 10 shows the bandwidth consumption. Several observations are in order. First, since all applications are prefetch-friendly (i.e., prefetcher has very high coverage as shown in Figure 10(a)), prefetching provides significant performance improvement regardless of the DRAM scheduling policy. The demand-prefetch-equal policy significantly outperforms the demand-first policy (by 28% in weighted speedup) because prefetches are very accurate in all applications.



(a) Individual speedup (b) System performance

Figure 9. Performance of a prefetch-friendly 4-core workload



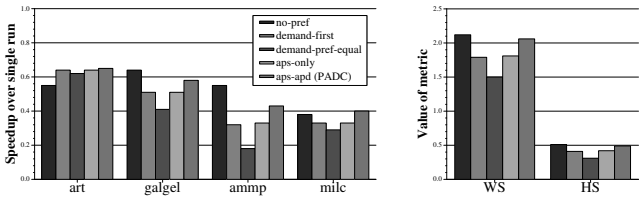
(a) Bus traffic breakdown (b) Total traffic

Figure 10. Bus traffic of a prefetch-friendly 4-core workload

Second, 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 increasing DRAM throughput, and 2) drops useless prefetches in *leslie3d* and *soplex*, thereby reducing their negative effects on all applications. Consequently, PADC improves prefetch coverage from 56% to 73% (as shown in Figure 10(b)). This is because it reduces contention for memory system resources by dropping useless prefetches from *leslie3d* and *soplex* which allows more useful prefetches to enter the memory system.

Finally, the bandwidth savings provided by PADC are 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 PADC can improve both performance and bandwidth-efficiency even when all applications benefit significantly from prefetching.

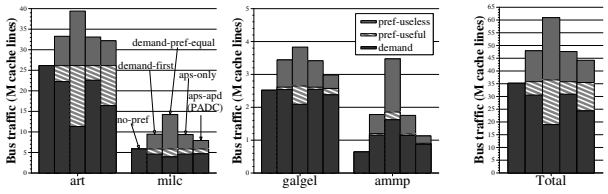
6.2.2. Case Study II: Prefetch-Unfriendly Applications We examine the behavior of PADC when four prefetch-unfriendly applications (*art, galgel, ammp*, and *milc*) are run together. Figures 11 and 12 show the performance and bandwidth consumption. Since the prefetcher is very inaccurate for all applications, prefetching degrades performance regardless of the scheduling policy. The demand-first and APS policies 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 APD drastically reduces the useless prefetches in all four applications (see Figure 12(a)) and therefore frees up memory system resources to be used by demands and useful prefetch requests. As a result, PADC outperforms the best previous prefetch scheduling policy for *all* applications.



(a) Individual speedup (b) System performance

Figure 11. Performance of a prefetch-unfriendly 4-core workload

PADC improves system performance by 17.7% (weighted speedup) and 21.5% (hmean speedup), while reducing bandwidth consumption by 9.1% over the baseline demand-first scheduler. By largely reducing the negative effects of useless prefetches both in scheduling

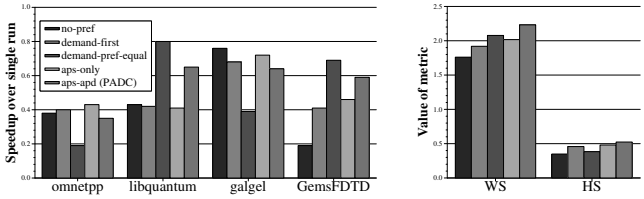


(a) Bus traffic breakdown (b) Total traffic

Figure 12. Bus traffic of a prefetch-unfriendly 4-core workload

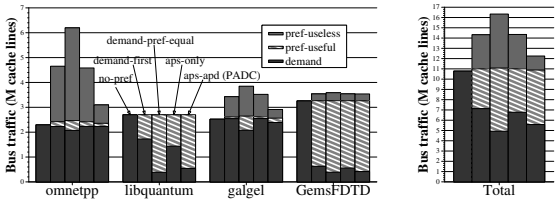
and memory system buffers/resources, PADC almost eliminates the system performance loss observed in this prefetch-unfriendly mix of applications. Weighted speedup and hmean speedup is within 2% and 1% of those obtained with no prefetching. We conclude that PADC can effectively eliminate the negative performance impact caused by inaccurate prefetching by intelligently managing the scheduling and buffer management of prefetch requests even in workload mixes where all applications are prefetch-unfriendly.

6.2.3. Case Study III: Mix of Prefetch-Friendly and Prefetch-Unfriendly Applications Figures 13 and 14 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-pref-equal significantly improves weighted speedup. However, the prefetcher generates many useless prefetches for *omnetpp* and *galgel* as shown in Figure 14(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 13. Performance of a mixed 4-core workload



(a) Bus traffic breakdown (b) Total traffic

Figure 14. Bus traffic of a mixed 4-core workload

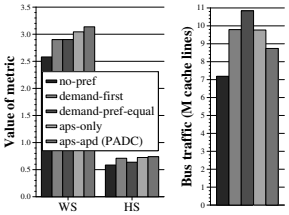
6.2.4. Effect of Prioritizing Urgent Requests We analyze the effectiveness of prioritizing urgent requests using the application mix in Case Study III. We say that a multi-core system is *fair* if each application experiences the same individual speedup. We use the unfairness metric (UF) in [17, 18] to indicate the degree of unfairness. Table 6 shows individual speedups, unfairness, WS, and HS for the workload for three policies: demand-first, a version of PADC that does not use the concept of "urgent requests," and PADC.

	Individual speedup				UF	WS	HS
	omnetpp	libquantum	galgel	GemsFDTD			
demand-first	0.404	0.425	0.681	0.410	1.687	1.919	0.458
PADC-no-urgent	0.206	0.936	0.418	0.696	4.547	2.256	0.410
PADC	0.352	0.649	0.636	0.594	1.845	2.231	0.524

Table 6. Effect of prioritizing urgent requests on perf. and fairness

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 6. In addition, harmonic speedup significantly improves at the cost of very little WS degradation.[10] This trend holds true for most workload mixes that consist of prefetch-friendly and prefetch-unfriendly applications. On average, prioritizing urgent requests improves HS by 8.8% for the 32 4-core workloads. We conclude that the concept of urgency significantly improves system fairness while keeping system performance high.

6.2.5. Overall Performance Figure 15 shows the average system performance and bus traffic for all 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 by 8.2% compared to both the demand-first and demand-prefetch-equal policies and reduces the bus traffic by 10.1% over the best-performing demand-first policy.



(a) Performance (b) Traffic

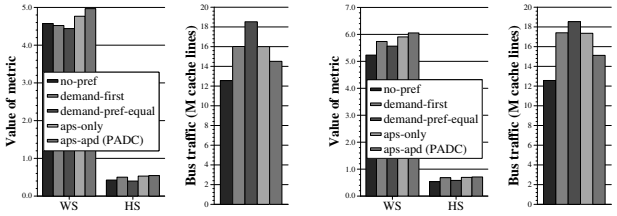
Figure 15. Results for 32 workloads on the 4-core system

We found that PADC outperforms both the demand-first and demand-pref-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*).

6.3. 8-Core Results

Figure 16(a) shows average performance and bus traffic over the 21 workloads we simulated on an 8-core system with a single memory controller and a single memory channel. Note that the rigid prefetch scheduling policies actually cause stream prefetching to degrade performance in the 8-core system. This is because DRAM bandwidth per core becomes a lot more scarce since the increased number of cores puts more pressure on the memory system. Also, resource contention and interference between prefetch and demand requests increase. For the very same reasons, PADC becomes more effective. As resource contention becomes higher, the benefit of intelligent prefetch prioritization and dropping of useless prefetches increases. PADC improves overall system performance (WS) by 9.9% on the 8-core system while also reducing memory bandwidth consumption by 9.4%. We expect the benefits of PADC will increase as off-chip memory bandwidth becomes a bigger performance bottleneck in future many-core systems.

[10] We found that, in many cases, prioritizing urgent requests improves weighted speedup as well.



(a) Single memory controller (b) Dual memory controllers

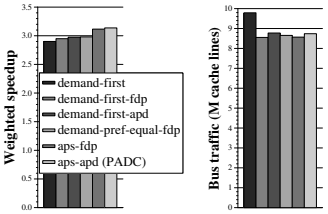
Figure 16. Results for 21 workloads on the 8-core system

Figure 16(b) evaluates the performance impact of PADC when two DRAM controllers are used in the 8-core system. Each controller works independently through a dedicated channel, doubling the peak memory bandwidth. Due to the increased bandwidth, the dual-controller system significantly reduces the contention between prefetch and demand requests and improves system performance over the single-controller system (WS improves by 30.9%). However, PADC still remains very effective with two memory controllers, improving WS by 5.5% and reducing bandwidth consumption by 13.2% compared to the previous-best demand-first policy. Therefore, we conclude that PADC is effective even on high-bandwidth DRAM systems.

6.4. Comparison with Feedback Directed Prefetching

Feedback Directed Prefetching (FDP) [28] adaptively adjusts the aggressiveness of the prefetcher in order to reduce its negative effects. We implemented FDP and tuned its parameters (prefetch accuracy, lateness, and pollution thresholds) for the stream prefetcher in our CMP system.

FDP is orthogonal to APS. As such, it can be employed together with APS. However, the benefits of FDP and APD overlap. FDP eliminates useless prefetches by reducing the aggressiveness of the prefetcher, which reduces 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 two advantages over FDP. First, FDP can be very slow in increasing the aggressiveness of the prefetcher when a new phase of execution starts. In such cases, FDP cannot issue useful prefetches whereas APD would have issued them because it always keeps the prefetcher aggressive. Second, FDP requires the tuning of multiple threshold values [28] to throttle the aggressiveness of the prefetcher and also has a greater hardware and parameter optimization cost than APD.



(a) Performance (b) Traffic

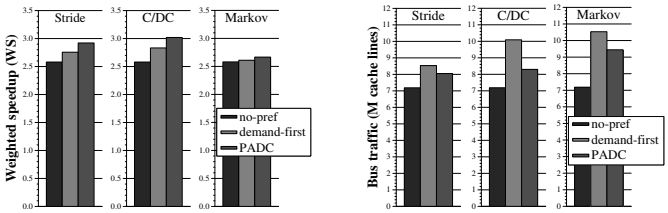
Figure 17. PADC vs. FDP

Figure 17 shows the performance and bus traffic of different combinations of FDP and PADC mechanisms averaged across the 32 workloads run on the 4-core system. When used with the demand-first policy, FDP improves performance by 1.7% and reduces bus traffic by 12.6% while APD improves performance by 2.9% and reduces bus traffic by 10.4%. Since FDP reduces the aggressiveness of the prefetcher, it is able to eliminate more useless prefetches (hence the lower bus traffic). However, for the very same reason, FDP eliminates some useful prefetches as well and therefore its performance improvement is not as high as that of APD.

When used together with APS, FDP improves performance by 7.4%. Hence, our adaptive scheduling policy and FDP are orthogonal and improve performance significantly when combined together. However, PADC outperforms the combination of FDP and APS, which shows that adaptive prefetch dropping is better suited to eliminate the negative effects of prefetching than feedback directed prefetching.

6.5. Effect on Other Prefetching Mechanisms

We briefly evaluate the effect of PADC on different types of prefetchers: PC-based stride [1], CZone/Delta Correlation (C/DC) prefetcher [22], and the Markov prefetcher [7]. Figure 18 shows the performance and bus traffic results averaged over all 32 workloads run on the 4-core system. PADC consistently improves performance and reduces bandwidth consumption compared to the demand-first policy and demand-prefetch-equal policy (not shown) with all prefetchers. We conclude that PADC is effective with a wide variety of prefetching mechanisms.



(a) Performance (b) Traffic

Figure 18. PADC on stride, C/DC, and Markov prefetchers

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. 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 [34, 24, 6], multi-threaded [20, 32, 6], and stream-based [13, 31] systems. In addition, several recent works [21, 17, 18] proposed techniques for fair DRAM scheduling across different applications sharing the DRAM system. Some of these previous proposals [34, 20, 32, 21, 17, 18, 6] do not discuss how prefetch requests are handled with respect to demand requests. Our mechanism is orthogonal to these scheduling policies: they can be extended to adaptively prioritize between demand and prefetch requests and to adaptively drop useless prefetch requests.

Other DRAM controller proposals take two different approaches to handling prefetch requests. Some proposals [24, 11, 5, 28] always prioritize demand requests over prefetch requests. Other proposals [31] 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. [16] proposed prefetch filtering mechanisms using on-chip caches. 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. [15] proposed a mechanism that cancels software prefetches when the prefetch issue queue is full. This mechanism is not aware of the usefulness of prefetches. In contrast, PADC drops a prefetch request only if its age is greater than a dynamically adjusted threshold (based on prefetch accuracy). PADC can be used with software prefetching to efficiently schedule and drop software prefetches. Srinivasan et al. [29] use a profiling technique to mark load instructions that are likely to generate useful prefetches. This mechanism needs ISA support to mark 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 [33] propose a mechanism that eliminates a prefetch request for an address if the prefetch request for the same address was useless in the past. In an extended version of this paper [10], we show that PADC outperforms their hardware filter by 6.5% on our 4-core system since their technique aggressively removes too many useful prefetches.

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-performance than these proposals, as described below.

Hur and Lin [5] designed a probabilistic prefetching technique that adjusts prefetcher aggressiveness. They also schedule prefetch requests to DRAM adaptively based on a count of the number of demand requests that cannot issue due to a DRAM bank conflict caused by a prefetch request. 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. [28] 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 6.4, 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 as well as with four different prefetchers. 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.

Acknowledgments

Many thanks to Minsik Cho, Viji Srinivasan, José A. Joao, Eiman Ebrahimi and other HPS members. The initial study on APD was performed when Chang Joo Lee worked as an intern for IBM. Chang Joo Lee and Veynu Narasiman were supported by IBM and NVIDIA Ph.D. fellowships respectively during this work. We also gratefully acknowledge the support of the Cockrell Foundation and Intel Corporation.

References

[1] J. Baer and T. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, 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] J. D. Gindele. Buffer block prefetching method. *IBM Technical Disclosure Bulletin*, 20(2):696–697, July 1977.

[4] I. Hur and C. Lin. Adaptive history-based memory scheduler. In *MICRO-37*, 2004.

[5] I. Hur and C. Lin. Memory prefetching using adaptive stream detection. In *MICRO-39*, 2006.

[6] E. Ipek, O. Mutlu, J. Martínez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA-35*, 2008.

[7] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *ISCA-24*, 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. 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. Technical Report TR-HPS-2008-002, University of Texas at Autin, 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*, 2001.

[13] S. A. McKee, W. A. Wulf, J. H. Aylor, M. H. Salinas, R. H. Klenke, S. I. Hong, and D. A. B. Weikle. Dynamic access ordering for streamed computations. *IEEE Transactions on Computers*, 49:1255–1271, Nov. 2000.

[14] Micron. *Datasheet: 2Gb DDR3 SDRAM, MT41J512M4 - 64 Meg x 4 x 8 banks*.

[15] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS-V*, 1992.

[16] 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.

[17] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO-40*, 2007.

[18] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA-35*, 2008.

[19] 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.

[20] C. Natarajan, B. Christenson, and F. Briggs. A study of performance impact of memory controller features in multi-processor server environment. In *WMPI*, 2004.

[21] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO-39*, 2006.

[22] K. J. Nesbit, A. S. Dhodapkar, J. Laudon, and J. E. Smith. AC/DC: An adaptive data cache prefetcher. In *PACT-13*, 2004.

[23] 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.

[24] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA-27*, 2000.

[25] A. J. Smith. Cache memories. *Computing Surveys*, 14(4):473–530, 1982.

[26] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. In *ASPLOS-9*, 2000.

[27] L. Spracklen and S. G. Abraham. Chip multithreading: opportunities and challenges. In *HPCA-11*, 2005.

[28] 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.

[29] 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, 1999.

[30] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46:5–25, Jan. 2002.

[31] C. Zhang and S. A. McKee. Hardware-only stream prefetching and dynamic access ordering. In *ICS-14*, 2000.

[32] Z. Zhu and Z. Zhang. A performance comparison of DRAM memory system optimizations for SMT processors. In *HPCA-11*, 2005.

[33] 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.

[34] 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.