Improving Memory Bank-Level Parallelism in the Presence of Prefetching

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

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

§Computer Architecture Laboratory (CALCM)
Carnegie Mellon University
onur@cmu.edu

ABSTRACT

DRAM systems achieve high performance when all DRAM banks are busy servicing useful memory requests. The degree to which DRAM banks are busy is called DRAM Bank-Level Parallelism (BLP). This paper proposes two new cost-effective mechanisms to maximize DRAM BLP. BLP-Aware Prefetch Issue (BAPI) issues prefetches into the on-chip Miss Status Holding Registers (MSHRs) associated with each core in a multi-core system such that the requests can be serviced in parallel in different DRAM banks. BLP-Preserving Multi-core Request Issue (BPMRI) does the actual loading of the DRAM controller's request buffers so that requests from the same core can be serviced in parallel, minimizing the serialization of each core's concurrent requests. When combined, BAPI and BPMRI improve system performance by 11.7% on a 4-core CMP system for a wide variety of multiprogrammed workloads. BAPI and BPMRI also complement various existing DRAM scheduling and prefetching algorithms, and can be used in conjunction with them.

Categories and Subject Descriptors: C.1.0 [Processor Architectures]: General; C.5.3 [Microcomputers]: Microprocessors;

General Terms: Design, Performance.

1. INTRODUCTION

Modern DRAM chips consist of multiple banks that can be accessed independently. Requests to different DRAM banks can proceed concurrently. As a result, their access latencies can be overlapped and DRAM throughput can improve leading to high system performance. The notion of servicing multiple requests in parallel in different DRAM banks is called *DRAM Bank-Level Parallelism (BLP)*.

Many sophisticated performance improvement techniques such as prefetching, out-of-order execution, and runahead execution [4, 16] are designed to amortize the cost of long memory latencies by generating multiple outstanding memory requests with the hope of exploiting Memory-Level Parallelism (MLP) [8]. The effectiveness of these techniques critically depends on whether the application's outstanding memory requests are actually serviced in parallel in different DRAM banks. If the requests in the DRAM controller's buffers (which we call DRAM request buffers) are NOT to different banks, the amount of BLP exploited will be very low, thereby reducing the effectiveness of such techniques.

This paper shows that the issue policy of memory requests into Miss Status Holding Registers (MSHRs) and DRAM request buffers significantly affects the level of BLP exploited by the DRAM controller. MSHRs [11] keep track of all outstanding cache misses for a processing core. All memory requests must first be allocated an MSHR entry before entering the DRAM request buffers where they are considered for DRAM scheduling. The request remains in the MSHR until serviced by DRAM. The MSHR structure is complex and not scalable in size [26] since it requires content-associative search. Therefore, the choice of which requests are placed into the MSHRs and finally into DRAM request buffers significantly affects the amount of BLP exploited by the DRAM controller. To this end, we propose new request issue policies into MSHRs and DRAM request buffers that aim to maximize DRAM BLP, and evaluate the effectiveness of our techniques on single and multi-core systems in the presence of two commonly-used mechanisms that exploit MLP: prefetching and out-of-order execution.

We propose two techniques: *BLP-Aware Prefetch Issue (BAPI)* and *BLP-Preserving Multi-core Request Issue (BPMRI)*. BAPI tries to maximize the BLP of prefetch requests exposed to the DRAM controller. It does so by prioritizing prefetch requests to different banks over prefetch requests to the same bank when issuing requests into the MSHRs. When BAPI is employed on each core of a CMP system, the increased BLP it exposes for a core can be destroyed by interference from other cores' requests. To prevent this, we propose BPMRI, which issues memory requests into DRAM request buffers such that the requests from each core (application) can be serviced by the DRAM controller without destroying the BLP of each application. BPMRI achieves this by issuing memory requests from the same core back-to-back to the DRAM request buffers.

Our evaluations show that the proposed mechanisms improve system performance significantly by increasing DRAM BLP. BAPI improves the performance of the 14 most memory intensive SPEC CPU 2000/2006 benchmarks by 8.5% on a single-core processor compared to a conventional memory system design. BAPI combined with BPMRI improves system performance (weighted speedup) for 30 multiprogrammed workloads by 11.7% on a 4-core CMP. We show that our mechanisms are simple to implement and low-cost, requiring only 11.5KB of storage in a 4-core CMP system.

Contributions: To our knowledge, this is the first paper that proposes adaptive memory request issue policies in the on-chip memory system that aim to maximize DRAM BLP. We make the following contributions:

1. This paper shows that BLP-unaware request issue policies in the on-chip memory system can destroy DRAM BLP, thereby significantly reducing the effectiveness of techniques that benefit from MLP such as prefetching and out-of-order execution.
2. We propose a new adaptive prefetch issue policy into MSHRs that increases the level of DRAM BLP exposed to the memory controller for an individual application.
3. We propose a new request issue policy to DRAM request buffers that tries to preserve the DRAM BLP of each application when multiple applications run together on a CMP system.
4. We evaluate our techniques for various prefetching algorithms and configurations on single-core and CMP systems, and show that they significantly improve system performance. We

MICRO'09, December 12–16, 2009, New York, NY, USA.

show that the proposed techniques complement parallelism- and prefetch-aware DRAM scheduling policies.

2. BACKGROUND AND MOTIVATION

2.1 Baseline CMP Memory System Design

Figure 1 illustrates our baseline CMP system design that consists of multiple cores and multiple DRAM channels. Each core has its own hardware prefetcher that monitors its L2 demand access stream to generate prefetch requests. Once generated, prefetch requests are buffered in a FIFO (First-In First-Out) buffer which we call the *prefetch request buffer*. This buffer is similar to the prefetch buffer for the L1 cache in the Intel Core processor [3].[1] The oldest prefetch in the prefetch request buffer is chosen to be sent to the *MSHR allocator* every processor cycle. The MSHR allocator decides whether an L1 instruction/data miss or a prefetch request is allocated. It prioritizes demands over prefetches since delaying the service of demands can hurt performance. Before an MSHR entry is allocated, all existing MSHRs are searched for a matching entry. If no matching entry is found, a free MSHR entry is allocated and the request is sent to the *L2 access buffer*. If this request is a prefetch, it is invalidated from the prefetch request buffer.

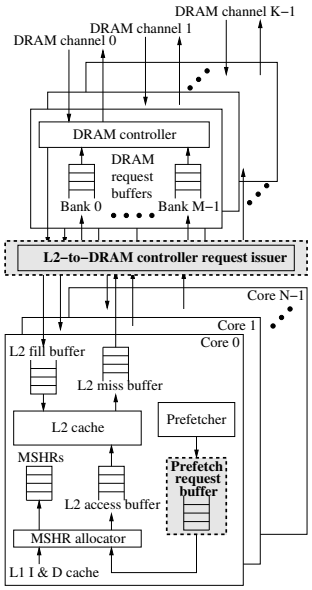


Figure 1: CMP memory system

When an L2 access (either a demand or prefetch) turns out to be an L2 miss, it is sent to the *L2 miss buffer* where it waits until it is sent to the DRAM request buffer in the corresponding DRAM channel. The destination (i.e., which DRAM request buffer in which DRAM channel) is predetermined based on the physical address of the request. L2 miss requests in the L2 access buffer from each core contend for *DRAM request buffers*. The *L2-to-DRAM Controller (L2-to-DC) request issuer* performs this arbitration in a round-robin fashion among cores. The DRAM controller in each DRAM channel examines all requests in its *DRAM request buffers* and decides which request will be serviced by the DRAM system. Once a request is serviced, the data from DRAM is sent to the L2 cache through the *L2 fill buffer* and the corresponding MSHR entry is freed.

The total number of outstanding demand/prefetch requests in the system cannot be more than the total number of MSHR entries. Also, the size of the DRAM request buffers limit the scope of the DRAM controller for DRAM access scheduling. Therefore, the order we send requests to the MSHRs and DRAM request buffers can significantly affect the amount of DRAM BLP exploited by the DRAM controller. In this paper, we focus on the issue policies for entrance into these two buffers. The parts of the baseline design we modify are highlighted in Figure 1.

[1]The FIFO queue in Intel's processor sends the oldest request to the L1 cache as long as a *Fill Buffer* entry (MSHR entry) is available. If the prefetch FIFO is full in Intel Core, a new prefetch overrides the oldest prefetch. Our prefetch request buffer is also a FIFO, but is connected to the L2 cache (instead of L1) and does not allow a new prefetch to override the oldest one (instead we just stall the prefetcher) since we found that overriding hurts performance by removing many useful prefetches.

2.2 Hardware Prefetchers

A hardware prefetcher speculates on an application's memory access patterns and sends memory requests to the memory system earlier than the application demands the data. If done well, the prefetched data is installed in the cache and future demand accesses that would have otherwise missed now hit in the cache. If the prefetcher generates a large number of useful prefetches, then significant performance improvement can be achieved.

We use a stream prefetcher [23, 12] similar to the one in IBM's POWER 4 [25] for most of our experiments. Stream prefetchers are commonly used in many processors [25, 9] since they do not require significant hardware cost and work well for many applications. Our implementation of the stream prefetcher is best-performing on average among a variety of prefetchers we evaluated for the 55 SPEC CPU 2000/2006 benchmarks. We also evaluate our mechanisms with other prefetchers in Section 5.3.

2.3 Prefetching: Increasing Potential for DRAM Bank-Level Parallelism

Hardware prefetchers can increase the potential for DRAM BLP because they generate multiple memory requests within a short period of time. With prefetching enabled, demand requests and potential future requests (useful prefetches) are both in the memory system at the same time. This increase in concurrent requests provides more potential to exploit DRAM BLP as shown in the following example.

Figure 2(a) shows a code example from *libquantum* where a significant number of useful prefetches are generated by the stream prefetcher. For ease of understanding, we abstract away many details of the DRAM system (also for Figures 3 and 5). Figure 2(b) shows the memory accesses generated when the code is executed both with and without a prefetcher. We assume that the first two accesses (to cache line addresses A, and A+1) are mapped to the same DRAM bank and that the two subsequent accesses (to A+2, and A+3) are mapped to a different bank.

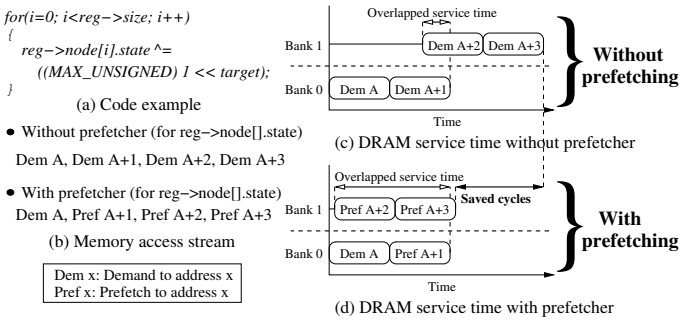


Figure 2: How prefetching can increase DRAM BLP (*libquantum*)

Figure 2(c) shows the DRAM service time when the code is executed without prefetching. Due to the lookahead provided by the processor's instruction window, accesses to A+1 and A+2 are slightly overlapped. On the other hand, with the prefetcher enabled, if the prefetches reach the memory system (DRAM request buffers) quickly such that the DRAM controller can see all these requests, the DRAM service time of the prefetches significantly overlap as shown in Figure 2(d). Therefore, overall DRAM service time is significantly improved compared to no prefetching (shown as "Saved cycles" in the figure).

As shown in the example, a hardware prefetcher can increase the potential for improving DRAM bank-level parallelism. However, we found that this potential is NOT always fully exposed to the DRAM system.

2.4 What Can Limit Prefetching's Benefits?

If an on-chip memory system design does not take DRAM BLP into account, it may limit the benefits of prefetching. For

example, the FIFO buffer (which we call prefetch request buffer) in the Intel Core design [3] buffers prefetch requests until they can be sent to the memory system. This FIFO structure will always send the oldest prefetch request to the memory system provided that the memory system has room for an additional request. This design choice can limit the amount of DRAM BLP exploited when servicing the prefetch requests since the oldest request in the buffer is always sent first regardless of whether or not it can be serviced in parallel with other requests. A more intelligent policy would consider DRAM BLP when sending prefetch requests to the memory system.

Figure 3 illustrates this problem. Figure 3(a) shows the initial state of the prefetch request buffer, MSHRs (three entries), and DRAM request buffers (three entries per DRAM bank). There is only one outstanding demand request (request 1 in the figure). This request is mapped to bank 0 and just about to be scheduled to access DRAM. There are five prefetches in the prefetch request buffer. The first two prefetches will access DRAM bank 0 and the three subsequent prefetches will access DRAM bank 1. For this example we assume that all the prefetches are useful and therefore will be required by the program soon.

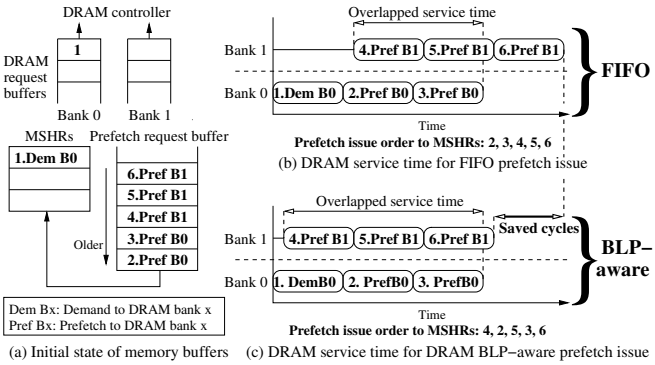


Figure 3: FIFO vs. DRAM BLP-aware prefetch issue policy

Figure 3(b) shows the DRAM service timeline when prefetches are issued into MSHRs in a FIFO fashion. In this case, the demand request and the two prefetch requests to bank 0 fill up the MSHRs and therefore the first prefetch to bank 1 will not be issued until the demand request gets serviced by DRAM and its MSHR entry is freed. As a result, BLP is low.

A DRAM BLP-aware issue policy would send a prefetch to bank 1 first, followed by a prefetch to bank 0. In other words, we can alternately issue prefetches to bank 1 and bank 0. Using this issue policy, the service of prefetches to bank 1 can start earlier and overlap with accesses to bank 0 as shown in Figure 3(c). Therefore, BLP increases and overall DRAM service time improves (shown as "Saved cycles" in the figure).

This example provides two insights. First, *simply increasing the number of outstanding requests in the memory system does not necessarily mean that their latencies will overlap.* A BLP-unaware prefetch issue policy (to MSHRs) can severely limit the BLP exploited by the DRAM controller. Second, a simple prefetch issue policy that is aware of which bank a memory request will access can improve DRAM service time by prioritizing prefetches to different banks over prefetches to the same bank.

So far we assumed that all prefetches are useful. However, if prefetches are useless, the BLP-aware prefetch issue policy will not be helpful. It may increase DRAM throughput but only for useless requests. We address this issue in Section 3.1.

3. MECHANISM

Our mechanism consists of two techniques to increase DRAM BLP. One is BLP-Aware Prefetch Issue (BAPI) which attempts to increase BLP for prefetch requests on each core. The other is BLP-Preserving Multi-core Request Issue (BPMRI) which tries to minimize the destructive interference in the BLP of each application when multiple applications run together on a CMP system. We describe these two techniques in detail below.

3.1 BLP-Aware Prefetch Issue

BLP-Aware Prefetch Issue (BAPI) tries to send prefetches from the prefetch request buffer to the MSHRs such that the number of different DRAM banks the requests access is maximized rather than sending the prefetches based on FIFO order. To achieve this, the FIFO prefetch request buffer is modified into the structures shown in Figure 4. Instead of having one unified FIFO buffer for buffering new prefetch requests before they enter MSHRs, BAPI contains multiple FIFOs (one per DRAM bank) that buffer new prefetch requests. However, to keep the number of supported new prefetch requests the same as the baseline and also to minimize the total storage cost dedicated to prefetch requests, we use multiple *index buffers* (one per DRAM bank) and a single, unified *prefetch request storage* structure. An index buffer stores indexes (i.e., pointers) into the prefetch request storage structure. The prefetch request storage structure is a regular memory array that stores prefetch addresses generated by the prefetcher. Last, there is a *free list* that keeps track of free indexes in the prefetch request storage structure. The index buffers and free list are all FIFO buffers and all of the buffers have the same number of entries as the baseline unified FIFO.

When the prefetcher generates a request, the free list is consulted. If a free index exists, the request address is inserted into the prefetch request storage structure at the index allocated to it. At the same time, that index is also inserted into the appropriate index buffer corresponding to the bank the prefetch is mapped to. BAPI selects one index among the oldest indexes from each index buffer every processor cycle. Then, the corresponding prefetch request (i.e., prefetch address) is obtained from the prefetch request storage and sent to the MSHR allocator. If the MSHR allocator successfully allocates an entry for the prefetch request, the selected index is inserted into the free list and also removed from the index buffer.

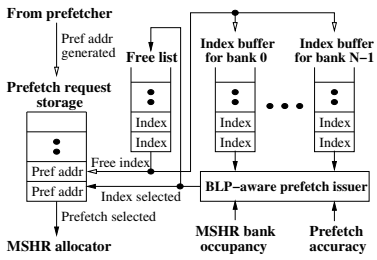


Figure 4: BLP-Aware Prefetch Issue

Prefetch Issue Policy: BAPI, shown in Figure 4, decides which prefetch to send to the MSHR allocator among the prefetch indexes from each index buffer. It makes its decision based on the DRAM BLP currently exposed in the memory system. To monitor the DRAM BLP of requests, the processor keeps track of the number of outstanding requests (both demands and prefetches) in the MSHRs separately for each DRAM bank. To accomplish this, we use a counter for each DRAM bank, called *MSHR bank occupancy counter*, which keeps track of how many requests to that bank are currently present in the MSHRs. When a demand/prefetch request is allocated an MSHR entry, its corresponding bank occupancy counter is incremented. When a request is serviced and its MSHR is freed, the corresponding bank occupancy counter is decremented.

The key idea of BAPI is to select the next prefetch to place into the MSHRs by examining MSHR bank occupancy counters such that the selected request improves the potential DRAM BLP. To do so, one would choose a prefetch request to the bank whose MSHR bank occupancy counter is the smallest. However, we found that this policy alone is not enough to expose more BLP to the DRAM controller for all applications. There are a large number of applications for which a prefetcher generates many prefetches to just a single bank but almost no prefetches to the other banks during a phase of execution (especially for streaming

applications). For such applications, the issue policy based on MSHR occupancy alone still ends up filling the MSHRs with requests to only one bank. This results in two problems. First, it results in no BLP improvement because the prefetches/demands to other banks that are soon generated cannot be sent to the memory system because the MSHRs are already full. Second, the MSHRs can be filled up with prefetches and thus demands that need MSHR entries can be delayed.

To prevent this problem, BAPI uses a threshold, $prefetch\_send\_threshold$ to limit the maximum number of requests to a single bank that can be outstanding in the MSHRs. This policy reserves room in the MSHRs for requests to other banks when most requests being generated are biased to just a few banks. Because many applications exploit row buffer locality in DRAM banks (since the access latency to the same row accessed last time is relatively low), having too low a threshold can hurt performance by preventing many of the useful prefetches to the same row from being row-hits (because the row may be closed before the remaining prefetch requests arrive). On the other hand, having too high a threshold will result in no BLP improvement as the MSHRs may get filled with accesses to only a few banks. Therefore, balancing the threshold is important for high performance. We empirically found that a value of 27 (when the total number of MSHR entries is 32) for $prefetch\_send\_threshold$ provides a good trade-off for SPEC benchmarks by exploiting BLP without constraining the row-buffer locality of requests.

Rule 1 summarizes our prefetch issue policy to MSHRs.

Rule 1 BLP-Aware Prefetch Issue policy (BAPI)

for each issue cycle **do**

1. Make the oldest prefetch to each bank *valid* only if the corresponding $MSHR\_bank\_occupancy\_counter$ value is less than $prefetch\_send\_threshold$.
2. Among those valid prefetches, select the request to the bank whose $MSHR\_bank\_occupancy\_counter$ value is least.

end for

Adaptive thresholding based on prefetch accuracy estimation: Prefetching may not work well for all applications or all phases of a single application. In such cases, performance improvement is low (or may even degrade) since useless prefetches will eventually be serviced, resulting in artificially high BLP and wasted DRAM bandwidth. A good prefetch issue policy should prohibit or allow sending prefetches to the memory system based on how accurate the prefetcher is. Our BLP-aware adaptive prefetch issue policy does exactly that: it limits the number of prefetches allowed in the MSHRs by dynamically adjusting $prefetch\_send\_threshold$ based on the run-time accuracy of the prefetcher. This naturally limits the number of prefetches sent to memory when prefetch accuracy is low. This improves performance for two main reasons: 1) It reserves more room in the MSHRs for demands, thereby reducing contention between demand requests and useless prefetches and 2) It effectively stalls the prefetcher from generating more useless prefetches since the prefetch request buffer will quickly become full.

To implement this, we need to measure the run-time accuracy of the prefetcher [23, 12, 5]. Therefore, we add an extra *prefetch bit* per L2 cache line and MSHR entry which indicates whether a line was brought in by a prefetch. Using this bit, each core keeps track of the number of useful prefetches in a counter, *useful prefetch counter*. It also counts the total number of prefetches in another counter, *prefetch sent counter*. Each core's prefetch accuracy is calculated by division of its two counters and the result is stored in its *prefetch accuracy register*. The two counters are reset over predetermined intervals so that the accuracy measurement adapts to the phases of an application.

BAPI dynamically adjusts $prefetch\_send\_threshold$ for each core based on the estimated prefetch accuracy. If the estimated accuracy is very low for an interval, a low $prefetch\_send\_threshold$ value is used which severely limits the number of useless prefetches sent to each bank. We empirically found that three levels of $prefetch\_send\_threshold$ work well for SPEC workloads. The threshold values used in our system are shown in Section 4.4.

3.2 BLP Preserving Multi-Core Request Issue

BLP-Aware Prefetch Issue (BAPI) increases the potential of DRAM BLP for individual applications on each core. In order for the DRAM controller to exploit this potential, the increased BLP should be exposed to the DRAM request buffers. However, in CMP systems, multiple cores share parts of the on-chip memory system. In our CMP system of Figure 1, the structures above the L2 miss buffers are shared by all cores. Therefore, requests from different cores contend for the shared DRAM request buffers in the DRAM controller. Due to this contention, a BLP-unaware L2-to-DRAM Controller (L2-to-DC) request issue policy can destroy the BLP of an individual application.

Figure 5 describes this problem. Figure 5(a) shows the initial state of the L2 miss buffers of two cores (A and B) and the DRAM request buffers for two banks. Each core has potential to benefit from BLP in that one request of each core goes to bank 0 and the other goes to bank 1. The L2-to-DC request issuer chooses a single request from the L2 miss buffers to be placed in the corresponding DRAM request buffer every cycle.[2]

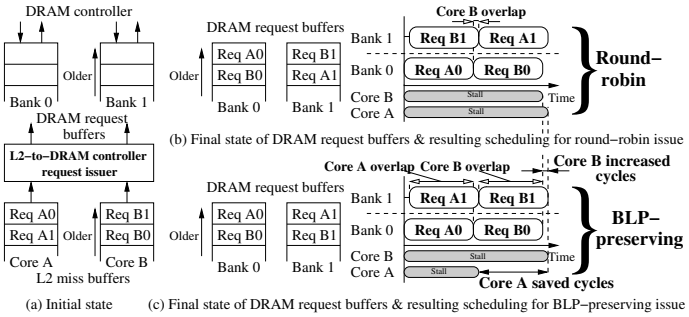


Figure 5: Round-robin vs. BLP-preserving request issue policy

Motivation: Existing systems use a round-robin policy in the L2-to-DC request issuer. Each cycle, a request from a different core is issued into DRAM request buffers and the cores are prioritized in a round-robin order. If such a policy is used as shown in Figure 5(b), core A's request to bank 0 is sent to the DRAM request buffers the first cycle and core B's request to bank 1 is sent the next cycle. The DRAM controller (based on the first-come first-served principle used in many existing systems [20, 24]) would service these requests (A0 and B1) from different cores concurrently because they are the oldest in each DRAM bank request buffer. This results in the destruction of the BLP potential of each core because requests from the same core are serviced serially instead of in parallel. Hence, the full latency of each request is exposed to each core and therefore each core stalls for approximately two DRAM bank access latencies.

On the other hand, a BLP-preserving L2-to-DC request issue policy would send all the requests from one core first as shown in Figure 5(c). Therefore, the DRAM controller will service core A's requests (A0 and A1) concurrently since they are the oldest in each bank. The requests from core B will also be serviced in parallel, after A's requests are complete. In this case, the BLP potential of each core is realized by the DRAM controller. The service of core A's requests finishes much earlier compared to the round-robin policy because core A's requests are overlapped. Core A stalls for approximately a single DRAM bank access

[2]The L2-to-DC request issuer need not run at the processor clock frequency since L2 misses are not frequent. In our evaluations, it runs at one-fourth the processor clock frequency.

latency instead of two and core B's stall time does not change much. Therefore, overall system performance improves because core A can make faster progress instead of stalling.

This example shows that a round-robin-based L2-to-DC request issue policy can destroy the BLP within an application by consecutively placing requests from different cores into the DRAM request buffers. As such, the DRAM controller may not be able to exploit the BLP potential of each application, which ultimately results in performance degradation. To ensure that each application makes fast progress with its DRAM requests serviced in parallel instead of serially, the L2-to-DC request issuer should preserve the BLP of requests from each core.

Mechanism: BLP Preserving Multi-core Request Issue (BPMRI) tries to minimize the destructive interference in the BLP of each application on a CMP system. The basic idea is to consecutively send many memory requests from one core to the DRAM request buffers so that the BLP of that core (or application) can be preserved in the DRAM request buffers for DRAM scheduling. If requests from a single core arrive consecutively (back-to-back) into the DRAM request buffers, they will be serviced concurrently as long as the requests span multiple DRAM banks, thereby preserving the BLP within the individual application. Note that our first technique, BAPI, already increases the likelihood that outstanding memory requests of a core are to different banks; hence, BAPI and BPMRI are synergistic.

BPMRI continues issuing memory requests from a single core into DRAM request buffers until the number of consecutive requests sent reaches a threshold, *request_send_threshold*, or there are no more requests in that core's L2 miss buffer. When this termination condition is met, BPMRI chooses another core and repeats the process. BPMRI selects the next core based on how memory intensive each application is. It prioritizes the core (application) that is the least memory intensive. To do this, BPMRI monitors the number of requests that come into the L2 miss buffer during predetermined intervals using a counter, *L2 miss counter*, for each core. At the start of an interval, BPMRI ranks each core based on the accumulated L2 miss counters (computed during the previous interval) and records the rank in a register, *rank register*, for each core. The core with the lowest value in its L2 miss counter is ranked the highest. The rank determined for each core is used to select the next core (upon meeting a termination condition) during that interval. The L2 miss counters are reset each interval to adapt to the phase behavior of applications. Rule 2 summarizes the BPMRI policy.

Rule 2 BLP-Preserving Multi-core Request Issue policy (BPMRI)

A *valid request* is a request in a core's L2 miss buffer that has a free entry in the corresponding bank's DRAM request buffer.

```
for each issue cycle do
  next core ← previous core
  cond1 ← no valid requests in next core's L2 miss buffer
  cond2 ← consecutive requests from next core >= threshold
  if cond1 OR cond2 then
    next core ← highest ranked core with valid request
  end if
  issue oldest valid request from next core
end for
```

We choose to limit the maximum number of consecutive requests sent and also choose to prioritize memory non-intensive applications since an uncontrolled "one core-first policy" can lead to the starvation of memory non-intensive applications. If a memory-intensive application continuously generates many requests, once those requests start to be issued into the DRAM request buffers, requests from other applications may not get a chance to enter the DRAM request buffers. Limiting the maximum number of requests consecutively sent from a single core alleviates this problem. In addition, the performance impact of delaying requests from a memory non-intensive application is more significant than delaying requests from a memory-intensive application. Therefore, prioritizing requests from memory non-intensive applications (ranking) leads to better overall system performance. Note that this approach is similar to the shortest-job-first policy in that it prioritizes shorter jobs (memory non-intensive cores that spend less time in the memory system) from the point of view of the memory system. The shortest-job-first policy was shown to lead to optimal system throughput [21].

4. METHODOLOGY

4.1 System Model

We use a cycle accurate x86 CMP simulator for our evaluations. Our simulator faithfully models all microarchitectural details such as bank conflicts, port contention, and buffer/queuing delays. The baseline on-chip memory system is modeled as shown in Figure 1. The baseline configuration of each core is shown in Table 1 and the shared resource configuration for single, 4, and 8-core systems is shown in Table 2. Our simulator also models a DDR3 DRAM system in detail and Table 3 shows the DDR3 DRAM timing specifications used for our evaluations.

Execution core	Out of order; decode/retire up to 4 instructions, issue/execute up to 8 microinstructions; 15 stages 256-entry reorder buffer; 32-entry MSHRs
Front end	Fetch up to 2 branches; 4K-entry BTB; 64K-entry gshare/PAs hybrid branch predictor
On-chip caches	L1 I and D: 32KB, 4-way, 2-cycle, 1 read/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: 32 stream entries, prefetch degree of 4, prefetch distance of 64 [25, 23], 128-entry prefetch request buffer

Table 1: Baseline configuration of each core

DRAM and bus	800MHz DRAM bus cycle, DDR3 1600MHz [14], 8 to 1 core to DRAM bus frequency ratio; 8B-wide data bus per channel, BL = 8; 1 rank, 8 banks per channel, 8KB row buffer per bank;
DRAM controllers	On-chip, open-row, demand-first [12] FR-FCFS [20] 1, 2, 4 channels for 1, 4, 8-core CMPs;
DRAM request buffers	64-entry (8 × 8 banks) for single-core processor 256 and 512-entry (16 × 8 banks per channel) for 4 and 8-core CMPs

Table 2: Baseline shared resource configuration

Param	DRAM cycles	Param	DRAM cycles	Param	DRAM cycles
^tRP	11	^tRCD	11	CL	11
CWL	8	AL	0	^tBL	4
^tRC	39	^tRAS	28	^tRTP	4
^tBL	4	^tCCD	4	^tRRD	4
^tFAW	24	^tWTR	4	^tWR	12

Table 3: DRAM timing specifications

4.2 Metrics

To measure CMP system performance, we use *Individual Speedup (IS)*, *Weighted Speedup (WS)* [22], and *Harmonic mean of Speedups (HS)* [13]. WS corresponds to system throughput and HS corresponds to the inverse of job turnaround time [6]. In the following equations, N is the number of cores in the CMP system. IPC_i^{alone} is the IPC when application i runs alone on one core of the CMP system (other cores are idle). $IPC_i^{together}$ is the IPC when application i runs on one core and other applications run on the other cores of the CMP system.

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

To evaluate how our mechanisms improve the performance of prefetching, we define prefetch-related metrics. *Bus traffic* is the number of cache lines transferred over the bus during the

		No prefetcher			Prefetcher							No prefetcher			Prefetcher				
Benchmark	Type	IPC	MPKI	BLP	IPC	MPKI	BLP	ACC	COV	Benchmark	Type	IPC	MPKI	BLP	IPC	MPKI	BLP	ACC	COV
171.swim	FP00	0.29	27.58	2.60	0.61	10.81	3.58	99.95%	60.79%	178.galgel	FP00	1.05	12.62	3.78	0.93	11.53	3.35	23.98%	12.50%
179.art	FP00	0.14	130.80	1.25	0.13	106.74	1.60	46.76%	18.40%	183.equake	FP00	0.48	19.89	1.29	1.08	0.78	1.89	94.76%	96.06%
189.lucas	FP00	0.48	10.61	1.60	0.62	3.01	1.60	72.81%	71.62%	429.mcf	INT06	0.12	39.08	1.86	0.13	36.03	1.98	23.00%	11.13%
410.bwaves	FP06	0.58	18.71	1.56	1.25	0.08	1.69	99.96%	99.57%	433.milc	FP06	0.40	29.33	1.40	0.35	21.13	1.94	20.24%	27.96%
437.leslie3d	FP06	0.46	21.14	1.64	0.76	2.06	2.20	88.25%	90.39%	450.soplex	FP06	0.36	21.52	1.37	0.64	3.58	1.84	81.83%	83.40%
459.GemsFDTD	FP06	0.42	16.29	2.27	0.81	1.95	2.80	90.36%	88.04%	462.libquantum	INT06	0.45	13.51	1.01	1.03	0.00	1.19	99.98%	99.99%
470.lbm	FP06	0.36	20.16	2.12	0.40	7.46	1.91	92.37%	63.01%	471.omnetpp	INT06	0.39	11.47	1.46	0.39	9.89	1.77	11.40%	19.84%

Table 4: Characteristics for 14 memory-intensive SPEC benchmarks with/without stream prefetcher: IPC, MPKI (L2 misses Per 1K Instructions), DRAM BLP, ACC (prefetch accuracy), and COV (prefetch coverage)

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:

$$ACC = \frac{Number\ of\ useful\ prefetches}{Number\ of\ prefetches\ sent},$$

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

We define DRAM *BLP* as the average number of DRAM banks which are busy (servicing a request) when at least one bank is busy. To evaluate the effect of DRAM throughput improvement on each 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.

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

4.3 Workloads

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. We ran each benchmark with the reference input set for 200 million x86 instructions selected by Pinpoints [19] as a representative portion of each benchmark.

The characteristics of the 14 most memory-intensive SPEC benchmarks with and without a stream prefetcher are shown in Table 4. To evaluate our mechanism on CMP systems, we formed combinations of multiprogrammed workloads from all the 55 SPEC 2000/2006 benchmarks. We ran 30 and 15 pseudo-randomly chosen workload combinations[3] for our 4 and 8-core CMP configurations respectively.

4.4 Implementation and Hardware Cost

For evaluations of BAPI, we use $prefetch\_send\_threshold$ values based on the run-time prefetcher accuracy as shown in Table 5. We use a value of 10 for *request_send_threshold* for BPMRI. The estimation of prefetch accuracy and rank recording is performed every 100K processor cycles. These values were empirically determined by simulations.

Prefetch accuracy (%)	0 - 40	40 - 85	85 - 100
$prefetch\_send\_threshold$	1	7	27

Table 5: Dynamic $prefetch\_send\_threshold$ values for BAPI

Table 6 shows the storage cost for our implementation of BAPI and BPMRI. The total storage cost for the 4-core system described in Tables 1 and 2 is 94,440 bits (∼11.5KB), which is equivalent to only 0.6% of the L2 cache data storage. Note that the additional FIFOs (for index buffers and free lists) and prefetch bits account for 99% of the total storage. FIFOs are made of regular memory arrays and index registers (pointers to the head/tail) and therefore the actual design cost/effort is not expensive. Prefetch bits are already used in many proposals [7, 29, 23, 12] to indicate whether or not a cache line (or request) was brought in (or made) by the prefetcher.

None of the issuing logic for BAPI or BPMRI is on the critical path of execution. Therefore, we believe that our mechanism is easy to implement with low design cost/effort.

	Structure	Cost equation (bits)	Cost for 4-core
BAPI	Index buffer	$N_{core} \times N_{channel} \times N_{bank} \times N_{buffer} \times log_2 N_{buffer}$	57,344
	Free list	$N_{core} \times N_{buffer} \times log_2 N_{buffer}$	3,584
	MSHR bank occupancy counter	$N_{core} \times N_{channel} \times N_{bank} \times (log_2 N_{MSHR} + 1)$	384
	Prefetch bit	$N_{core} \times (N_{line} + N_{MSHR})$	32,896
	Prefetch sent counter	$N_{core} \times 16$	64
	Prefetch used counter	$N_{core} \times 16$	64
	Prefetch accuracy register	$N_{core} \times 8$	32
BPMRI	L2 miss counter	$N_{core} \times 16$	64
	Rank register	$N_{core} \times log_2 N_{core}$	8
Total storage cost for the 4-core system in Table 1 and 2			94,440
Total storage cost as a fraction of the L2 cache capacity			**0.6%**

Table 6: Hardware storage cost of BAPI and BPMRI (N_{line}, N_{core}, N_{MSHR}, N_{buffer}, $N_{channel}$, N_{bank}: number of L2 cache lines, cores, MSHR entries, prefetch request buffer entries, DRAM channels, DRAM banks per channel)

5. EXPERIMENTAL EVALUATION

5.1 Single-Core Results

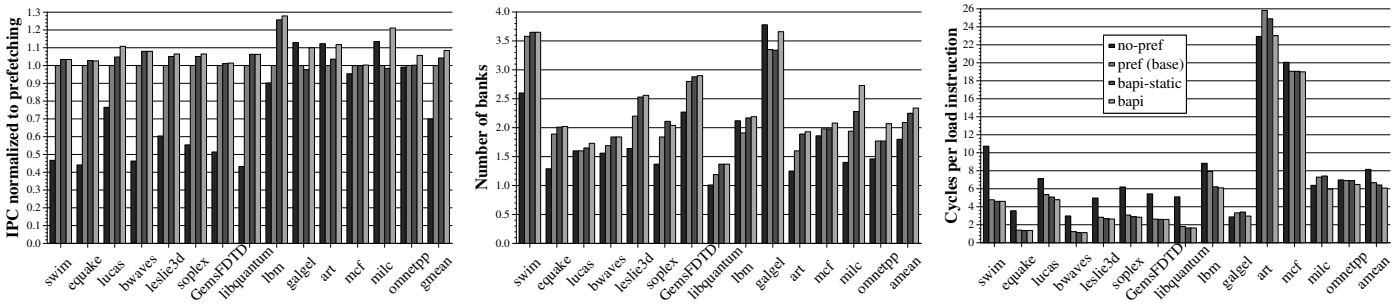
We evaluate BLP-Aware Prefetch Issue (BAPI) in this section. Recall that BAPI aims to increase the BLP potential of a single application whether the application is running alone on a single core machine or running together with other applications on a CMP system. To eliminate the effects of inter-application interference, we first evaluate BAPI on our single core system.

Figures 6 and 7 show IPC, DRAM BLP, stall cycles per load instruction (SPL), and bus traffic for the 14 most memory intensive benchmarks when we use 1) no prefetching, 2) the baseline with stream prefetching (using the FIFO prefetch issue policy), 3) BAPI with a static threshold (BAPI-static), and 4) BAPI (with adaptive thresholding; BAPI-dynamic or simply BAPI). BAPI-static uses a single constant value for *prefetch_send_threshold* which is set to 27 empirically, whereas BAPI-dynamic varies this threshold based on the accuracy of the prefetcher (as shown in Table 5). IPC is normalized to prefetching with the baseline issue policies.

On average, BAPI-dynamic improves performance over the baseline by 8.5%. This improvement is due to two major factors: 1) increased DRAM BLP of prefetches in phases where the prefetcher works well, and 2) limiting the issue of prefetches for applications or phases where the prefetcher is inaccurate. These two factors are analyzed in detail below.

Analysis: Both BAPI-static and dynamic improve performance for the nine leftmost benchmarks shown in Figure 6(a). These benchmarks are all prefetch friendly as can be seen in Figure 7: most of the prefetches are useful (high prefetch accu-

[3]We imposed the requirement that each of the multiprogrammed workloads have at least one memory-intensive application since these applications are most relevant to our study. We consider an application to be memory-intensive if its L2 Misses Per 1K Instructions (MPKI) is greater than 5.



(a) Performance (b) DRAM BLP (c) Stall cycles per load instruction

Figure 6: Performance, BLP, and SPL of BAPI on single-core system

racy) and these useful prefetches cover a majority of the total bus traffic (high prefetch coverage).

BAPI increases performance over baseline prefetching by exposing more DRAM BLP of prefetches to the DRAM controller. As shown in Figure 6(b), BAPI increases BLP for these nine applications and therefore improves DRAM throughput. This leads to significant reductions in stall cycles per load (SPL) as shown in Figure 6(c). DRAM throughput improvement also leads to high prefetch coverage. Since MSHR entries are freed sooner due to better DRAM throughput, more prefetches are able to enter the memory system which improves prefetcher coverage. This is best illustrated by the increase in useful prefetches with BAPI for *swim* and *lbm* as shown in Figure 7.

Note that for *lbm*, baseline prefetching with FIFO issue degrades DRAM BLP while improving performance by 10.9% compared to no prefetching. *lbm* consists of multiple sequential memory access streams in a loop and therefore it exploits DRAM BLP even without prefetching. The stream prefetcher is beneficial by bringing in many cache lines earlier than needed; hence, it improves performance. However, this is done in a BLP inefficient way due to the FIFO prefetch issue policy as described in Section 2.1. In other words, the FIFO prefetch issue policy significantly limits the DRAM BLP potential for *lbm* by filling up the MSHRs with prefetch requests that span just a few banks even though there are many younger prefetches to other free DRAM banks waiting in the prefetch request buffer. As a result, the prefetcher's performance improvement is relatively small compared to the other prefetch friendly benchmarks. BAPI mitigates this problem by prioritizing prefetches to different banks, thereby improving DRAM BLP by 15.1% and overall performance by 27.9% compared to the FIFO issue policy.

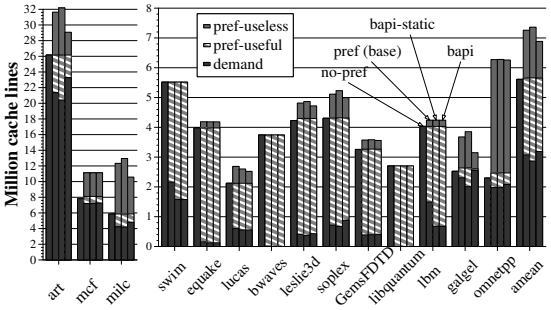


Figure 7: Bus traffic of BAPI on single-core system

Adaptivity to Usefulness of Prefetches: On the other hand, for the five rightmost benchmarks, BAPI-static does not improve performance over the baseline. As shown in Figure 7, the stream prefetcher does not work well for these benchmarks: it generates a large number of useless prefetches which unnecessarily consume on-chip buffer/cache resources and DRAM bandwidth. As shown in Figure 6(a), prefetching degrades performance for *galgel, art*, and *milc*. BAPI-static does not help these benchmarks either since the useless prefetches are still serviced. In fact, for *galgel, art*, and *milc*, BAPI-static increases the number of useless prefetches due to increased DRAM throughput as shown in Figure 7. Thus, BLP-aware prefetch issue alone does not help performance when prefetch accuracy is low.

BAPI-dynamic alleviates the problem of useless prefetches by limiting the number of prefetches issued into the MSHRs when the prefetcher generates a large number of useless prefetches. As a result, MSHR entries do not quickly fill up with useless prefetches and thus can be used by demand requests. This mechanism causes the prefetch request buffer to fill up, thereby stalling the prefetcher. As shown in Figure 7, BAPI-dynamic eliminates a large number of useless prefetches and reduces total bus traffic by 5.2% on average. BAPI-dynamic almost recovers the performance loss due to useless prefetches for *galgel* and *art*, and improves performance for both *milc* and *omnetpp* by 6.6%.

Adaptivity to Phase Behavior: BAPI-dynamic adapts to the phase behavior of *lucas, leslie3d, soplex, GemsFDTD*, and *lbm*. While most of the time the prefetcher generates useful requests, in certain phases of these applications it generates many useless prefetches. BAPI-dynamic improves performance for these benchmarks by adaptively adjusting $prefetch\_send\_threshold$ which removes many of the useless prefetches while keeping the useful ones as shown in Figure 7.

We conclude that BAPI significantly improves performance (by 8.5%) by increasing DRAM BLP (by 11.7%) while also reducing memory bus traffic (by 5.2%) in the single-core system.

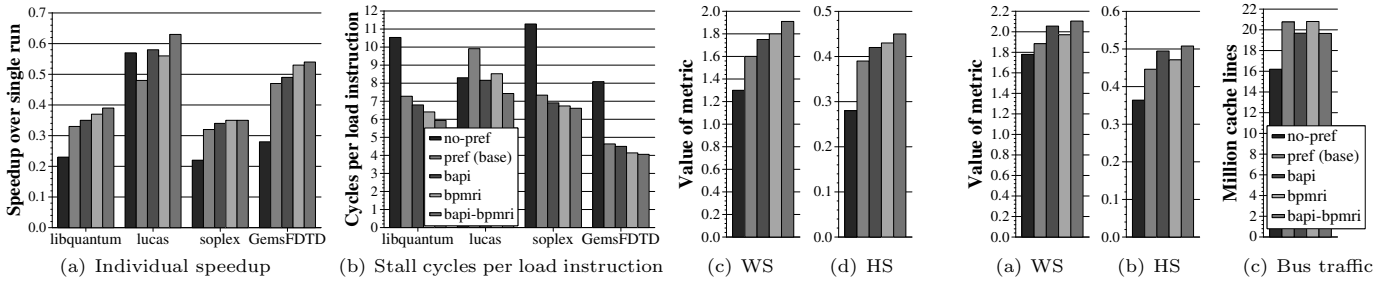
5.1.1 Sensitivity to MSHR Size

Thus far we have assumed that each core has a limited number of MSHR entries (32) because MSHRs are not scalable since they require associative search [26]. In this section, we study the effect of our techniques with various MSHR sizes. We varied the total number of MSHR entries from 8 to 256 and measured the average IPC (gmean) for the 14 most memory-intensive benchmarks as shown in Table 7. To isolate the effect of limited MSHRs, we assume that there is an unlimited number of DRAM request buffer entries for this experiment (this is why the IPC improvement of BAPI with a 32-entry MSHR is different from that shown in Section 5.1). The values of $prefetch\_send\_threshold$ are empirically determined for both BAPI-static and BAPI separately for each MSHR size to provide the best performance.

MSHR entries	8	16	32	64	128	256
Storage cost	0.6KB	1.3KB	2.5KB	5.1KB	10.1KB	20.3KB
no-pref IPC	0.36	0.38	0.38	0.38	0.38	0.38
pref (base) IPC	0.43	0.50	0.53	0.56	**0.59**	0.58
bapi-static IPC	0.47	0.54	0.57	0.59	0.59	0.58
bapi IPC	0.48	0.55	**0.59**	0.60	0.61	0.61
bapi-static's IPC Δ	8.5%	9.1%	7.8%	4.0%	0.0%	-0.1%
bapi's IPC Δ	10.5%	10.3%	10.0%	6.4%	3.0%	4.3%

Table 7: Average IPC performance with various MSHR sizes

We make three major observations. First, as the number of MSHR entries increases, the performance of baseline prefetching increases since more BLP is exposed in DRAM request buffers.



(a) Individual speedup (b) Stall cycles per load instruction (c) WS (d) HS

Figure 8: Case study on the 4-core system (*libquantum, lucas, soplex,* and *GemsFDTD*)

(a) WS (b) HS (c) Bus traffic

Figure 9: Performance on 4-core CMP

The performance improvement saturates at 128 entries because the DRAM system itself becomes the performance bottleneck when a high level of BLP is exposed. In fact, increasing the MSHR size from 128 to 256 entries slightly degrades performance because more useless prefetches of some applications (especially, *art* and *milc*) enter the memory system (due to the large number of MSHR entries) causing interference with demand requests both in DRAM and in caches.

Second, both BAPI-static and BAPI (with dynamic thresholding) continue to improve performance up to 64-entry MSHRs since they expose more BLP of prefetches to DRAM request buffers. Even though BAPI-static's performance saturates at 64 MSHR entries, BAPI improves performance with 128 and 256-entry MSHRs because it continues to expose higher levels of *useful* BLP without filling the memory system with useless prefetches. Its ability to adaptively expose useful BLP to the memory system and thereby more efficiently utilize the MSHR entries makes BAPI best-performing regardless of MSHR size.

Finally, BAPI with a smaller MSHR achieves the benefits of a significantly larger MSHR without the associated cost of building one: BAPI with 32-entry MSHRs performs as well as the baseline with 128-entry MSHRs. Similarly, BAPI with 16-entry MSHRs performs within 1% of the baseline with 64-entry MSHRs. Note that BAPI requires very simple extra logic and FIFO structures (∼2KB storage cost for the single-core system) whereas increasing the number of MSHR entries is more costly in terms of both latency and area due to two reasons [26]: 1) MSHRs require associative search, 2) MSHRs require the storage of cache line data. We conclude that BAPI is a cost-effective mechanism that efficiently uses MSHRs and therefore provides higher levels of BLP without the cost of large MSHRs.

5.2 Multi-Core Results

In this section, we evaluate BLP-Aware Prefetch Issue (BAPI) and BLP-Preserving Multi-core Request Issue (BPMRI) when employed together in CMP systems. To provide insight into how our mechanisms work, we begin with a case study.

5.2.1 Case Study on the 4-Core System

We evaluate a workload consisting of four prefetch-friendly (high prefetch accuracy and coverage) applications to show how our mechanisms further improve the benefits of prefetching and thus system performance by improving and preserving DRAM BLP. Figure 8 shows performance metrics when *libquantum, lucas, soplex,* and *GemsFDTD* run together on the 4-core system.

As shown in Figure 8(c), prefetching with the baseline issue policies (FIFO prefetch issue and round-robin L2-to-DC request issue) improves WS by 23.5% compared to no prefetching. This increase is due to the performance improvement of *libquantum, soplex*, and *GemsFDTD*. The performance of *lucas* actually degrades even though baseline prefetching improves performance for *lucas* on the single-core system. There are two reasons for this. First, the baseline round-robin L2-to-DC issue policy destroys the BLP of requests for *lucas* the most among the four applications. Since *lucas* is the least memory intensive (as shown in Table 4) of the four applications, the issue of *lucas*'s requests to DRAM request buffers is relatively infrequent compared to the others. As a result, 1) *lucas*'s requests starve behind more intensive applications' requests in the L2 miss buffer and 2) *lucas*'s BLP is more easily destroyed because requests from other applications intervene between *lucas*'s requests when a round-robin issue policy is used. Second, although amenable to prefetching in general, the prefetch accuracy of *lucas* is not as good compared to the other applications, and therefore *lucas* suffers the most from useless prefetches (as shown in Section 5.1).

BPMRI alleviates the first problem as shown in Figures 8(a) and (b). BPMRI ranks *lucas*'s requests highest because *lucas* is the least memory intensive application among the four. Whenever BPMRI needs to choose the next core to issue requests from, *lucas* gets prioritized and its requests are issued consecutively into the DRAM request buffers. Therefore, *lucas*'s starvation is mitigated and its BLP is preserved. BPMRI regains the performance lost due to baseline prefetching as shown in Figure 8(a). BPMRI also significantly improves the performance of the other three benchmarks by preserving the BLP of each application, thereby improving WS and HS by 12.0% and 11.3% respectively compared to the baseline.

BAPI mitigates the second problem of *lucas*. As discussed in Section 5.1, BAPI adapts to the phase behavior of *lucas*: when the prefetcher generates many useless prefetches, BAPI limits the issue of prefetches thereby reducing many of the negative effects of prefetching. On the other hand, BAPI exposes more BLP of prefetches to the memory system when the prefetcher is accurate. Therefore, BAPI increases performance for *lucas* as well as the other three applications, improving WS and HS by 9.4% and 7.9% compared to baseline prefetching.

When BPMRI and BAPI are combined, the performance of each application further improves as each application's SPL is reduced as shown in Figure 8(b). BAPI increases each application's BLP potential and BPMRI preserves this BLP thereby allowing the DRAM controller to exploit it. As a result, WS and HS improve by 19.4% and 17.4% respectively compared to the baseline BLP-unaware request issue policies.

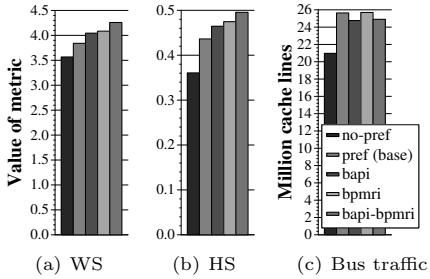
5.2.2 Overall Performance on the 4-Core System

Figure 9 shows the average system performance and bus traffic for all 30 workloads. When employed alone, BAPI improves average performance (WS) by 9.1%, BPMRI by 4.6% compared to the baseline. Combined together, BAPI and BPMRI improve WS and HS by 11.7% and 13.8% respectively, showing that the two techniques are complementary. Bus traffic is also reduced by 5.3%. The performance gain of the two mechanisms are due to 1) increased DRAM BLP provided by intelligent memory issue policies, 2) reduced waste in DRAM bandwidth and on-chip cache space due to limiting the number of useless prefetches.

5.2.3 Overall Performance on the 8-Core System

Figure 10 shows the average system performance and bus traffic for the 15 workloads we examined on the 8-core system. BAPI and BPMRI are still very effective and significantly improve sys-

tem performance. Combined together, they improve WS and HS by 10.9% and 13.6%, while reducing bus traffic by 2.9%. In contrast to the 4-core system where BAPI alone provided higher performance than BPMRI alone, BPMRI alone improves performance more than BAPI alone. This is because as the number of cores increases, destructive interference in each application's BLP also increases, and reducing this interference becomes a lot more important.



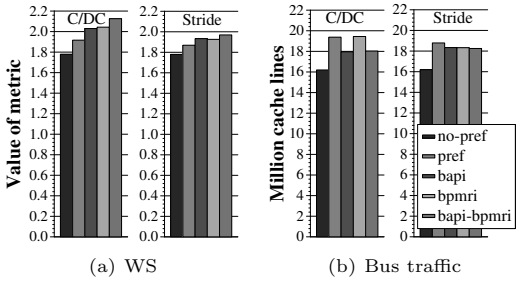
(a) WS (b) HS (c) Bus traffic

Figure 10: Performance for 8-core CMP

We conclude that the proposed techniques are effective in terms of both performance and bandwidth-efficiency for a wide variety of multiprogrammed workloads on both 4-core and 8-core systems.

5.3 Effect on Other Prefetchers

We evaluate our mechanisms on two different types of prefetchers: GHB (Global History Buffer)-based CZone Delta Correlation (C/DC) [18] and PC-based stride [1]. Both the C/DC and stride prefetchers accurately capture a substantial number of memory accesses that are mapped to different DRAM banks, just as the stream prefetcher does. Therefore, BAPI and BPMRI improve system performance compared to the baseline (WS: 10.9% and 5.4%, for C/DC and stride respectively). Our techniques also reduce bus traffic by 4.7% and 2.9% for C/DC and stride respectively. To conclude, our proposal is effective for a variety of state-of-the-art prefetching algorithms.



(a) WS (b) Bus traffic

Figure 11: BAPI and BPMRI with stride and C/DC prefetchers

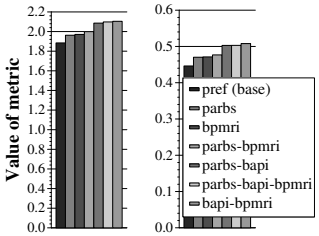
5.4 Comparison with Parallelism-Aware Batch DRAM Scheduling

Parallelism-Aware Batch Scheduling (PAR-BS) [15] aims to improve performance and fairness in DRAM request scheduling. It tries to service memory requests *in the DRAM request buffers* from the same core concurrently so that the DRAM BLP of each application is preserved in DRAM scheduling. Therefore the amount of BLP exploited by PAR-BS is limited by the number of requests to different banks in DRAM request buffers.

BAPI complements PAR-BS: it increases the number of prefetches to different banks and PAR-BS can exploit this increased level of BLP to improve performance further. BPMRI also complements PAR-BS even though their benefits partially overlap. If an application's requests to different banks are not all in the DRAM request buffers, PAR-BS cannot exploit the full BLP of each application. BPMRI, by consecutively issuing an application's requests from the L2 miss buffer to the DRAM request buffers, increases the probability that each application's requests to different banks are all in the DRAM request buffers. Hence, BPMRI increases the potential of each application's BLP that can be exploited by PAR-BS.

In addition, by consecutively issuing requests from a core back-to-back into the DRAM request buffers, BPMRI enables *any* DRAM controller to service those requests in parallel. Hence, a first-come-first-serve based DRAM controller combined with BPMRI can preserve each application's BLP without requiring the DRAM controller to be BLP-aware.

To verify this, we implemented PAR-BS tuned for best performance for our 4-core workloads. Figure 12 shows the performance of 1) baseline prefetching with the FR-FCFS DRAM scheduling policy which exploits row-buffer locality [20], 2) PAR-BS, 3) BPMRI, 4) PAR-BS with BPMRI, 5) PAR-BS with BAPI, 6) PAR-BS with BAPI and BPMRI, and 7) BAPI and BPMRI.



(a) WS (b) HS

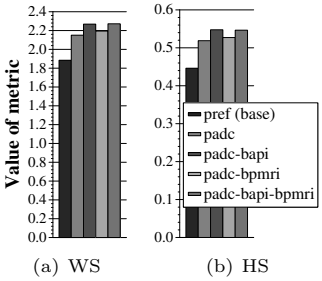
Figure 12: Comparison with PAR-BS

BPMRI's performance gain is equivalent to that of PAR-BS (with the round-robin L2-to-DC issue policy) since it successfully preserves the BLP of each application and makes the simple FR-FCFS DRAM scheduling policy behave similarly to PAR-BS. When combined with PAR-BS, BPMRI improves WS and HS by an additional 1.9% and 1.4% by better preserving the BLP of requests from each application. BAPI along with PAR-BS significantly improves the performance of PAR-BS (WS and HS improve by 7.1% and 7.3% respectively) because BAPI exposes more BLP potential of each application in the DRAM requests buffers for PAR-BS to exploit. To conclude, our mechanisms 1) complement PAR-BS, and 2) enable parallelism-unaware DRAM controllers to achieve similar performance as PAR-BS.

5.5 Comparison with Prefetch-Aware DRAM Controllers

Prefetch-Aware DRAM Controllers (PADC) [12] was proposed to maximize DRAM row buffer hits for useful requests (demands and useful prefetches). PADC also delays and drops useless prefetches to reduce waste in on-chip buffer resources and DRAM bandwidth. PADC aims to minimize DRAM latency of useful requests by prioritizing useful row-hit requests over others to the same bank. In other words, the main goal of PADC is to exploit row buffer locality in each bank in a useful manner. Our goal is orthogonal: BAPI and BPMRI *aim to maximize DRAM bank-level parallelism so that more requests from an application can be serviced in different DRAM banks in parallel.*

Figure 13 shows the performance of PADC alone and PADC combined with our mechanisms for the 4-core workloads. PADC significantly improves WS and HS by 14.1% and 16.3% respectively compared to the baseline. When combined with PADC, BAPI and BPMRI improve WS and HS by 20.6% and 22.5%. We conclude that our mechanisms complement PADC and thus significantly improve system performance.



(a) WS (b) HS

Figure 13: Comparison with PADC

6. RELATED WORK

6.1 DRAM Access Scheduling

A number of DRAM scheduling policies [20, 28, 17, 15, 12] have been proposed. Although these proposals have the similar goal of improving performance by increasing DRAM throughput, they do so by improving the DRAM controller's scheduling policy. Therefore, their scope is limited by the number and composition of requests in the DRAM request buffers. If the requests in the DRAM request buffers are not to different banks, BLP will be low regardless of the DRAM scheduling policy. Our mechanism solves this problem by issuing requests into on-chip buffers in a BLP-aware manner. It exposes more BLP to the DRAM scheduler, enabling it to provide higher DRAM throughput. As such, our techniques are orthogonal to DRAM scheduling policies. As shown in Sections 5.4 and 5.5, our mechanisms complement parallelism-aware and prefetch-aware DRAM scheduling.

6.2 Memory-Level Parallelism

Many proposals have explored increasing Memory-Level Parallelism (MLP) [8, 16, 27, 2, 26]. These works define MLP as the average number of outstanding memory requests when there is at least one outstanding request to memory. They implicitly assume that the DRAM latency of outstanding requests to memory will overlap. In contrast, we *show that simply having a large number of outstanding requests does not necessarily mean that their latencies will overlap.* In order to overlap, the requests should span multiple banks and be in the DRAM controller concurrently, which our mechanism enables. Hence, our proposal is orthogonal to and improves the effectiveness of techniques that improve MLP. As we quantitatively showed in this paper, our proposal provides significant benefits over two MLP-improving techniques, prefetching and out-of-order execution, by enabling them to better exploit BLP.

6.3 Prefetch Handling

Adaptive prefetch handling techniques [10, 23, 12, 5] aim to reduce the interference between prefetch and demand requests in the memory system. In contrast, our work focuses on increasing and preserving the DRAM BLP of useful requests (demands and useful prefetches) and therefore is orthogonal to prefetch handling mechanisms. As we discuss in Section 5.5, our mechanisms are complementary to prefetch-aware DRAM controllers [12] which employ an adaptive prefetch handling technique that is reported to outperform feedback-directed prefetching [23].

7. CONCLUSION

We showed that uncontrolled memory request issue policies to resource-limited on-chip buffers limit the level of DRAM bank-level parallelism (BLP) that can be exploited by the DRAM controller, thereby limiting system performance. To overcome this limitation, we proposed new cost-effective on-chip memory request issue mechanisms. Our evaluations show that the mechanisms 1) work synergistically and significantly improve both system performance and bandwidth-efficiency, 2) work well with various types of prefetchers, and 3) complement various DRAM scheduling policies. We conclude that our proposed mechanisms improve system performance and bandwidth efficiency for both single-core and multi-core systems and can be an enabler for achieving and enhancing the benefits of a multitude of techniques that are designed to exploit memory-level parallelism.

Acknowledgments

We thank Khubaib, other HPS members, and the reviewers for their comments. Chang Joo Lee was supported by an IBM scholarship 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] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *ISCA-31*, 2004.

[3] J. Doweck. Inside Intel Core microarchitecture and smart memory access. *Intel Technical White Paper*, 2006.

[4] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *ICS-11*, 1997.

[5] E. Ebrahimi, O. Mutlu, and Y. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *HPCA-15*, 2009.

[6] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3), 2008.

[7] J. D. Gindele. Buffer block prefetching method. *IBM Technical Disclosure Bulletin*, 20(2), July 1977.

[8] A. Glew. MLP yes! ILP no! In *ASPLOS Wild and Crazy Idea Session '98*, 1998.

[9] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 2001.

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

[11] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA-8*, 1981.

[12] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-aware DRAM controllers. In *MICRO-41*, 2008.

[13] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *ISPASS*.

[14] Micron. *2Gb DDR3 SDRAM, MT41J512M4 - 64 Meg x 4 x 8 banks*. http://download.micron.com/pdf/datasheets/dram/ddr3/.

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

[16] 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.

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

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

[19] 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.

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

[21] W. E. Smith. Various optimizers for single stage production. *Naval Research Logistics Quarterly*, 3, 1956.

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

[23] 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.

[24] Sun Microsystems, Inc. *OpenSPARC$^{(TM)}$ T1 Microarchitecture Specification*.

[25] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Technical White Paper*, 2001.

[26] J. Tuck, L. Ceze, and J. Torrellas. Scalable cache miss handling for high memory-level parallelism. In *MICRO-39*, 2006.

[27] H. Zhou and T. M. Conte. Enhancing memory level parallelism via recovery-free value prediction. In *ICS-17*, 2003.

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

[29] X. Zhuang and H.-H. S. Lee. Reducing cache pollution via dynamic data prefetch filtering. *IEEE TC*, 56(1), 2007.