Meeting Midway: Improving CMP Performance with Memory-Side Prefetching

Praveen Yedlapalli, Jagadish Kotra, Emre Kultursay, Mahmut Kandemir, Chita R. Das and Anand Sivasubramaniam
The Pennsylvania State University
{praveen, jbk5155, euk139, kandemir, das, anand}@cse.psu.edu

***Abstract*—Both on-chip resource contention and off-chip latencies have a significant impact on memory requests in large-scale chip multiprocessors. We propose a memory-side prefetcher, which brings data on-chip from DRAM, but does not proactively further push this data to the cores/caches. Sitting close to memory, it avails close knowledge of DRAM state and memory channels to leverage DRAM row buffer locality and channel state to bring data (from the current row buffer) on-chip ahead of need. This not only reduces the number of off-chip accesses for demand requests, but also reduces row buffer conflicts, effectively improving DRAM access times. At the same time, our prefetcher maintains this data in a small buffer at each memory controller instead of pushing it into the caches to avoid on-chip resource contention. We show that the proposed memory-side prefetcher outperforms a state-of-the-art core-side prefetcher and an existing memory-side prefetcher. More importantly, our prefetcher can also work in tandem with the core-side prefetcher to amplify the benefits. Using a wide range of multiprogrammed and multithreaded workloads, we show that this memory-side prefetcher provides IPC improvements of 6.2% (maximum of 33.6%), and 10% (maximum of 49.6%), on an average when running alone and when combined with a core-side prefetcher, respectively. By meeting requests midway, our solution reduces the off-chip latencies while avoiding the on-chip resource contention caused by inaccurate and ill-timed prefetches.**

Index Terms: memory prefetching, NOC, DRAM, CMP

I. Introduction

The growing scale of chip multiprocessors (CMPs) is magnifying the memory wall problem in several ways: (i) a larger number of hops has to be traversed in the on-chip network to get to the memory controllers before a request can even be issued to the memory (DRAM); (ii) the increased data demand from the cores causes higher contention for on-chip cache and network resources; (iii) higher contention at the memory controllers results in larger queuing delays; and (iv) the intermixing of requests across cores can lead to poor locality across pages in the row buffers of the DRAMs, thereby incurring higher latencies to switch the required rows before serving requests. All these detrimental factors accentuate the memory wall problem over and beyond the growing bandwidth disparity between the on-chip compute versus the off-chip memory.

One well-studied technique to address the memory wall problem is *prefetching*, both software [19], [27], [34] and hardware [12], [18], wherein memory requests are (predicted and) issued ahead of their need so that the corresponding instructions find the requested data in an appropriate level of the cache when they execute. The prefetch accuracy, in terms of both *what* and *when*, is a key determinant to hiding the memory latency. In fact, studies [31], [37] have shown that aggressive and inaccurate prefetches can even degrade performance. Further, prefetches in the context of large CMPs have additional ramifications. They can increase memory bandwidth requirements [31] and network traffic (i.e., lead to higher contention) [3], [6], and cause higher interference over cache lines [33], [37], worsening the performance even further. There have been recent proposals [5], [16], [31], [33] to reduce some of these effects in CMPs, though much of the prior work in this area has been in the context of "core-side" prefetching (a term broadly referring to prefetching that is initiated by the cores or caches). Even though core-side prefetching has the potential of having a more accurate knowledge of memory reference patterns (the *what* and *when*), a core-side prefetch request still experiences the same effects (i)-(iv) identified above as normal memory requests. Our experimental evaluation shows that, while a sophisticated core-side prefetching brings significant performance improvements in an 4-core system, it brings only meager improvements when we move to a 32-core system.

An earlier idea that has been explored in the context of uniprocessors is "memory-side" prefetching [8], [9], [12], [30], [35] where the memory system observes reference stream and pushes appropriate lines into the cache. This idea was primarily proposed to address the storage space and logic needed to maintain large on-chip predictor tables and/or to reduce the critical path of prefetch messages. We, however, believe that a memory side prefetcher can be leveraged from a very different perspective in the context of large CMPs, as explained below:

- With memory (DRAM) performance becoming a serious bottleneck, understanding and optimizing the requests issued to it based on its current state (especially leveraging the row buffer locality) is becoming extremely critical [1], [32], [36]. A memory-side prefetcher may have more accurate knowledge to optimize such locality (reduce row-buffer conflicts and further augment the effectiveness of memory controller scheduling algorithms [15], [26]), compared to a core-side prefetcher which does not have instantaneous access to memory state.
- While a core-side prefetcher relies on accurate predictions for being effective, a memory-side prefetcher can afford to be more "opportunistic", e.g., initiate certain requests based on bandwidth availability in the memory system. A core-side prefetcher is unaware of the memory status to nimbly adjust its aggressiveness.

This research is supported in part by NSF grants #1213052, #1152479, #1147388, #1139023, #1017882, #0963839, #0811687, #0953246, and grants from Intel and Microsoft Corporations.

978 -1-4799-1021-2/13/$31.00 ©2013 IEEE

• A core-side prefetcher employs *round-trip messages*, and its inaccuracies increase the contention in the on-chip network as will be shown in later. Its purpose is to bring the data into an appropriate cache/buffer so that hit rates can be improved. In a large-scale CMP having dozens of cores, while one would ideally want the data to be present in the caches when requested, *reducing miss latencies can be even more important*. Though one could build a general scheme (as in prior work [10], [13], [17], [28], [29]) where the data is pushed up to the caches, it does *not* need to be a mandatory requirement for a memory-side prefetcher. A memory-side prefetcher that just brings data on-chip (say to the memory controller), can cut as much as 82% of the total round-trip latency of a normal load request as shown in Table I. Consequently, unlike a core-side prefetcher, a memory-side prefetcher need not contribute to on-chip network contention as show in the next section, while still offering the potential of removing off-chip latency.

• We want to emphasize that a memory-side prefetcher does not preclude the provisioning of a core-side prefetcher as well. In fact, it is quite possible that the two could work in unison (as a decoupled prefetching solution), with the former bringing the data on-chip while leveraging the instantaneous state and capabilities of the memory banks/channels, and the latter leveraging the predictability of core requests to pick up the data (without going off-chip) brought in by the former.

Based on these ideas, this paper proposes a memory-side prefetcher that is integrated with the memory controllers of an on-chip network based CMP. It maintains a relatively small prefetch buffer of about 256KB per controller, into which our logic prefetches cache lines from DRAM row buffers based on various factors – predictability, bandwidth availability, row buffer conflict overheads, utility to different request streams, etc. We compare our memory-side prefetcher to a state-of-the-art stream-based core-side prefetcher which employs techniques from [5], a next-line core-side prefetcher, a variation of our memory-side prefetcher which pushes the data to caches, and an existing memory-side prefetcher [17] which pushes the data prefetched at bus idle times to on-chip caches. We show that our memory-side prefetcher outperforms all of those prefetchers. Using both *multiprogrammed* and *multithreaded* workloads, with varying memory pressures, running on a 32-core simulation platform with DDR3 memory, we show that our proposal gives an average IPC improvement of 6.2% (maximum of 33.6%) over no prefetching case when running alone, and 10% (maximum of 49.6%) when combined with a core-side prefetcher. We also perform a sensitivity study to demonstrate the robustness of our memory-side prefetcher in different configurations. Our results show that:

• Core-side prefetching does cut memory access latencies for CMPs, however its effectiveness decreases sharply as the CMPs get larger (like 32 cores).

• Existing memory-side prefetching schemes have better scope in large CMPs, but their effectiveness is limited due to increase in on-chip queueing delays.

• On large-scale CMPs, our mid-way memory-side prefetcher generates better results than conventional core-side and memory-side prefetchers, and complements core-side prefetcher to amplify the benefits.

To our knowledge, this is the first paper to propose a memory-side prefetcher for network-based CMPs, that strives to bring data **midway** (i.e., on-chip but not to the caches) *to avoid the detrimental on-chip effects of prefetching while alleviating/avoiding the off-chip latencies of demand requests*.[1]

II. Background and Motivation

In this section, we give an overview of the memory architecture in a state-of-the-art system; further details can be found in [4], [11], [23].

A. Background

In an on-chip network based CMP with S-NUCA [14] cache organization, upon an L1 miss, the L1 cache controller issues a request to the L2 cache bank corresponding to the missing address which traverses the on-chip network. If the address misses in the L2 cache as well, the L2 cache controller at that bank issues a request to the corresponding memory controller which again uses the on-chip network. The memory controller determines the memory bank corresponding to the requesting address and places it in the particular bank's queue. When it is time for the memory controller to schedule this request, corresponding memory commands are sent to the memory module (DRAM) over the memory channel. If the memory bank has the requesting address in its *row buffer* (*row buffer hit*), the response will be sent back to the memory controller. Otherwise, the row that is open must be closed and a new row must be activated (*row buffer conflict*) before the data can be read. This data response travels from the memory controller back to the requesting L2 cache controller through the on-chip network. Finally, the L2 cache controller forwards the response to the original requesting L1 cache.

B. Motivation

Workload	On-chip	Off-chip	
		Queueing	Access
High MPKI	18%	60%	22%
Moderate MPKI	35%	19%	46%
Low MPKI	43%	8%	49%

TABLE I. Breakdown of on-chip and off-chip latencies for L2 misses in three different workloads without any prefetching.

In a CMP system running multiple applications/threads, memory is a shared resource with high contention, which makes it one of the most important performance bottlenecks. Within a short period of execution, multiple L2 misses can happen and those requests will be sent to the memory controllers. To leverage memory level parallelism, multiple memory controllers and multiple banks at each memory controller are deployed. Still, any core in the system can access any bank at any memory controller. This flexibility leads to *inter-core interference* at the memory controllers. This contention/interference in the memory system can lead to significant queueing delays, thereby reducing overall memory system performance.

Table I shows the breakdown of the average round-trip latency incurred by a memory request in a 32-core CMP with 4 memory controllers in three different multiprogrammed workloads, prioritized from high (>10) MPKI (Misses Per Kilo Instructions) to low(<1) MPKI. In high and moderate MPKI workloads, the total off-chip latencies clearly dominate,

[1]The choice of prefetching algorithm is orthogonal to main contributions.

with a bulk of these latencies coming from the queueing delays at the memory controller in the high MPKI case. Note that while the DRAM access itself contributes directly to the latency of a memory request, it also indirectly manifests in the queueing delays of other requests waiting at the memory controller. Bringing data on-chip (as is done in our proposal) can reduce such off-chip accesses, which is especially useful in high MPKI applications. At the same time, note that in low MPKI workloads, where on-chip latencies become comparable to off-chip latencies, it is equally important to ensure that we do not add to the on-chip latencies when optimizing the off-chip costs. These contentions at the on-chip network and off-chip memory channels are becoming more prominent with increasing number of cores on chip and get further accentuated in large-scale CMPs if we simply introduce a conventional, core-side prefetcher, as depicted in Table II. *The results in this table indicate that a core-side prefetcher can cause significant queueing latency increase in both on-chip network and off-chip memory channels.* As we will show later, these high latencies are experienced due to ill-timed and inaccurate prefetches.

Queueing delay	Core-side prefetching	Memory-side prefetching
On-chip	22%	-
Off-chip	66%	-57%

TABLE II. PERCENTAGE INCREASE IN ON-CHIP (NETWORK) AND OFF-CHIP (MEMORY) QUEUEING DELAYS CAUSED BY A CONVENTIONAL CORE-SIDE ADAPTIVE STREAM PREFETCHING [18] AND OUR MEMORY-SIDE PREFETCHING WITH RESPECT TO "NO PREFETCHING".

These observations motivate our proposal for a memory-side prefetcher which attempts to bring some data from the currently active row in the DRAM on-chip to reduce the off-chip costs for accessing this data while leveraging the fact that proactively reading such data from the row buffer does not incur DRAM row switch out/in costs. Note that such prefetching does *not* need to push/propagate this data to the caches/cores, thus avoiding additional on-chip resource contention which can happen with normal inaccurate and ill-timed core-side prefetches. *Avoiding/reducing off-chip accesses, while not increasing on-chip traffic and not creating cache pollution serves as the primary motivation for our memory-side prefetcher.*

III. MEMORY-SIDE PREFETCHING

In this paper, we use memory-side prefetching to improve the latency of off-chip memory requests in large CMPs. Our prefetching scheme improves memory performance in two ways: (i) An access to prefetched (on-chip) data is served faster. (ii) An access to prefetched data will not use the memory bank or the memory channel, which reduces the contention on these highly shared resources, and as a result, reduces the queuing delay for all memory requests. We now explain the details of our proposed prefetching scheme.

A. What to Prefetch?

While prefetching has the potential to improve the performance of the memory system, it is not a light-weight task and comes with its own overheads. Therefore, accuracy of what to prefetch is extremely important. Our first criterion in identifying the data to be prefetched is that the data should come from an *open row* that is in the row buffer. A prefetch from a row buffer can be fast and does not disturb the current contents of the row buffer. Within a row, there are many cache lines that we can potentially prefetch. We analyzed the line access patterns of different applications, specifically focusing on the order in which the lines in a row are accessed after the row is opened.

The three graphs in Figure 1 plot the line access patterns of some representative applications. Each application is run individually on a single core platform with one memory controller to obtain these graphs. The xy-plane represents the line (each of size 64B) numbers within a row (of size 4KB), running from 0 to 63. For any point (x,y) in this plane, the value in the z-axis represents how frequently line y was accessed immediately after line x within the same row.

To quantify the line locality of an application, we define a metric called *next-line locality* (NLL), which is the average of all the values in the diagonal with offset 1 in the line access pattern graph (i.e., values at points $(i, i+1)$). This value represents the percentage of times a line is accessed immediately after its preceding line from the *same row*. Table V gives the NLL values of all the SPEC CPU2K6 applications. We see that the chances of accessing the next line in the same row after a line is accessed is about 36.8% over all applications. Extending the definition of next-line locality, *next-k-line locality* represents the percentage of times one of lines $(i+1, i+2, .., i+k)$ are accessed immediately after line i is accessed in the same row. The average next-k-line locality of our evaluated applications is around 43.7% for $k = 4$. This indicates that prefetching multiple subsequent lines may be a reasonable option in practice.

We selected next-line prediction for its simplicity to merely illustrate the benefits of memory-side prefetching and placing the data midway. Clearly, one could use a more sophisticated predictor like Adaptive Stream Detection [9]. The predictor for the prefetching scheme is in itself orthogonal to the main contributions of this paper. A sophisticated predictor has the potential to improve the accuracy of the memory prefetcher, further adding to the overall benefits one could achieve.

B. When to Prefetch?

It is of utmost importance to identify the "ideal time" to initiate a prefetch from the row buffer. The options are: (1) when a row buffer conflict happens, (2) when a row buffer hit happens, (3) when the row is first opened, or (4) when the memory bank and channel are idle.

1) Prefetch at Row Buffer Conflict:: In a system with the open page policy, when a row buffer conflict (RBC) happens, the row in the row buffer is precharged (closed) and a new row is activated (opened). Just before the row is closed, we prefetch the lines that are spatially proximate to the last accessed line from that row. We call this scheme *Prefetch at RBC*.

A potential drawback of this scheme is that, we are actually making the demand request wait for the prefetching to complete. The prefetch operation is started when the request is identified as a row buffer conflict and it has to be completed before the row is closed. Then a new row is opened to serve the demand request. This prefetching overhead falls into the critical path of the memory accesses. The latency of the row buffer conflicting requests increases significantly, thereby reducing overall memory performance. Another potential problem is that

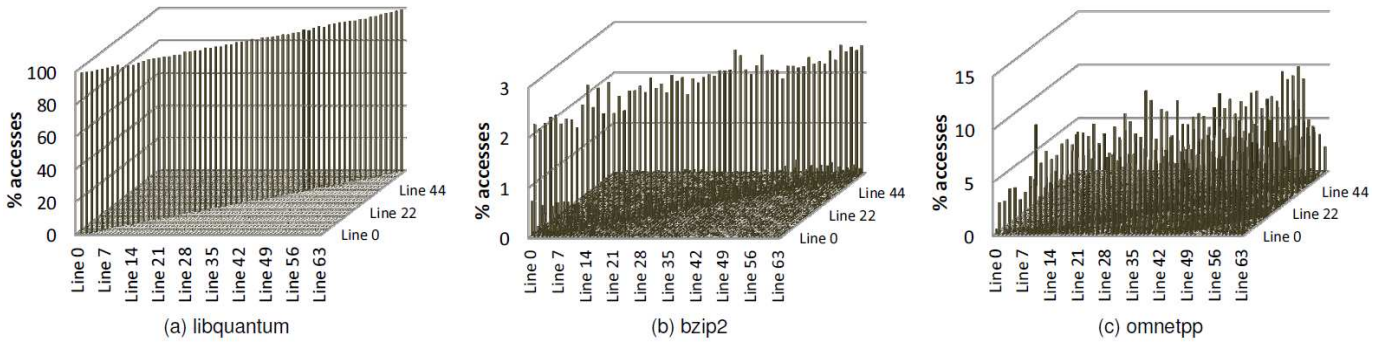

Fig. 1. Line access pattern graphs of some SPEC CPU2K6 applications.

issuing a prefetch at every row buffer conflict can result in a high number of prefetches being issued, causing a problem especially when such prefetches are not very useful.

2) Prefetch at Row Buffer Hit:: Another opportunity for prefetching data from the row buffer is upon a row buffer hit (RBH). Along with a demand request, we issue prefetch requests for the next lines from the row buffer. We call this scheme *Prefetch at RBH*. By doing the prefetching at row buffer hit, we are not making the demand-request wait. Instead, we first service the demand-request and then perform our prefetch. Subsequent requests to those lines in that row (which were supposed to be row buffer conflicts or hits) can now become prefetch hits. One more advantage of *Prefetch at RBH* is the prefetches may more accurately reflect sequentiality.

A prefetch hit is much faster than a row buffer hit as it hides the latency of accessing the memory and is served directly from an on-chip buffer. Compared to Prefetch at RBC, the number of prefetches issued with Prefetch at RBH will be lower because of fewer row buffer hits compared to the row buffer conflicts in a system with multiple applications. Note that we prefetch only at *read* row buffer hits since we are prefetching to serve only read requests.

3) Prefetch at Row Activation:: Another opportunity to prefetch from the row buffer is when a new row is activated. Similar to the *Prefetch at RBH* case, along with the demand-request, we can issue prefetch requests for the next lines from the row buffer. This scheme has the advantage of not putting the prefetching activity in the critical path of memory accesses. However, it has the same problems of *Prefetch at RBC* scheme regarding lack of locality and aggressive prefetching. As a consequence, we do not expect this scheme to perform better than the previous schemes, and therefore, do not pursue it any further.

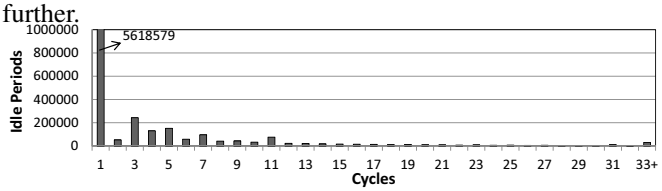

Fig. 2. Memory channel idle period length distribution under a moderate MPKI workload.

4) Prefetch at Idle Times:: Memory channel is a major bottleneck and the contention on the channel caused by prefetches can cause significant performance loss. Yet, we observed that the channel utilization in our baseline system without any prefetching is only about 30%. This means that there are lot of idle cycles on the channel. Our next prefetching scheme, called *Prefetch at Idle*, exploits this fact and issues prefetch requests when the channel is idle to an idle bank that got at least one row buffer hit. We further analyzed the channel idleness and looked at the distribution of the idle periods on the channel while running a moderate MPKI workload, shown in Figure 2. The x-axis represents the length of the idle period in cycles and y-axis represents the number of periods with that many cycles. From this figure, we see that most of the idle periods are very small (80% are 1-cycle), which are not useful for prefetching without penalizing demand requests. To complete the prefetching from an open row without affecting any other accesses, we need around 8 cycles on the channel (for a prefetch degree of 2). In the light of these results, this scheme is not expected to give good performance improvement. Still, in our experimental evaluation, we discuss a variant of this scheme (a multicore version of the single-core based memory-side prefetching scheme in [17]), and compare it with our proposal.

Prefetch Scheme	Critical Path	Locality	# of Prefetches
Prefetch at RBC	Yes	No	High
Prefetch at RBH	No	Yes	Low
Prefetch at Row ACT	No	No	High
Prefetch at Idle Times	No	Yes	High

TABLE III. CHARACTERISTICS OF DIFFERENT PREFETCH SCHEMES.

Table III gives a summary of the memory-side prefetching schemes discussed in this section. A good prefetching scheme should not be on the critical path, should consider locality before prefetching, and should not issue too many prefetches. Figure 3 compares different prefetch timing mechanisms quantitatively. It plots the percentage IPC improvements in various workloads[2] under different timing schemes compared to the *no prefetching* case. The first bar for each workload shows the IPC improvement in the case of *Prefetch at RBC*, and the second and third bars show the improvements in the cases of *Prefetch at RBH* and *Prefetch at Idle*. *Prefetch at RBH* scheme gave the highest IPC improvements whereas *Prefetch at RBC* actually degrades the IPC in many workloads because of the above explained reasons and *Prefetch at IDLE* scheme is in between the two. We conclude that *prefetching at row buffer hits* is the best way to minimize the prefetching overheads and improve accuracy. Therefore, we use this scheme in the rest of the paper.

[2]The description of our setup and workloads is given in Section V-B.

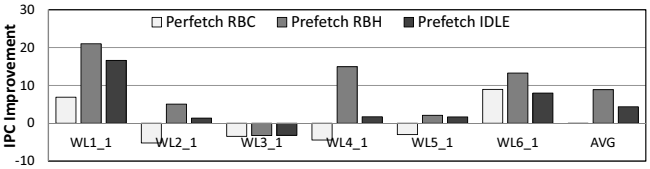

Fig. 3. Performance with different prefetch timing schemes.

C. Where to Prefetch?

Once the questions of *what to prefetch* and *when to prefetch* are answered, the next question that comes up is *where to store the prefetched data?* Prefetched data should be stored on-chip to avoid off-chip latencies. Typically, core-side prefetchers bring the prefetched data to on-chip caches. In this work, we explore the option of storing the prefetched data in a separate buffer on-chip called *Prefetch Buffer* in the memory controller.[3] As explained earlier, this cuts down a substantial part of the off-chip latency, while avoiding the problems of core-side prefetching.

The prefetch buffer is logically organized as a cache. Each row in this cache holds a prefetched memory block data and this data is identified uniquely by its corresponding memory address. The memory controller populates a new entry in the *prefetch buffer* when new data is prefetched from memory. Before issuing each memory read request, the memory controller does a lookup on the *prefetch buffer* and serves the request from the buffer if it is a hit.

We allocate a separate prefetch buffer at each memory controller. The prefetch buffer can be organized in two ways: (1) *shared prefetch buffer*, where any prefetch entry in the buffer can be used by any core, and (2) *private prefetch buffer*, where each core gets a specific set of prefetch entries in the buffer. A shared prefetch buffer suffers from the same problem of memory interference already observed by the memory controller queues. In this case, cores with high MPKI can take up more entries in the prefetch buffer, leading to fairness issues. Another problem with shared prefetch buffer is the extra cost of performing an associative search over a cache with relatively higher associativity. On the other hand, using private prefetch buffers reserves each core its own set of entries, and therefore, eliminates the interference between the cores at the prefetch buffer. While one can think that the data shared across applications or threads of a multithreaded application can lead to duplicate entries in the prefetch buffer which would not exist in a shared prefetch buffer, our experiments indicated that even with data sharing, the use of a shared prefetch buffer is not justified to offset the higher cost having a shared prefetch buffer. Therefore, we employ private per-core prefetch buffers.

D. Optimizations for Memory-Side Prefetching

Memory prefetching comes at the cost of extra pressure on the memory resources such as channels and banks. As a result, prefetching too frequently (e.g., along with every row buffer hit) can become an overkill. In some situations, serving a demand-request might be more beneficial than doing a prefetch (although prefetch will be done from an open row), because, while the prefetch is being done, all demand-requests in the bank queue get delayed. Further, application characteristics can vary dynamically throughout execution. There can be cases where an application or a phase of an application might not benefit from memory prefetching because of lack of locality. We implemented three optimizations to improve the effectiveness of the proposed memory-side prefetching.

- **Precharge on Prefetch:** After prefetching from an open row, there is a low chance of getting further immediate requests to that same row due to the filtering effect of the prefetch buffer. Consequently, we can precharge the row, thereby saving few cycles for subsequent requests to that bank.
- **Averting Costly Prefetches:** We do not issue prefetches when a high number of demand requests are queued, waiting for the channels or banks to become available. We calculate the total load on the memory bus and do prefetching only if the load is below a predetermined threshold (*Queueing Thresh*).
- **Prefetch Throttling:** We monitor the *usefulness* of prefetching for each application at runtime and reduce the prefetch degree dynamically for individual applications that do not benefit from it. For each application, the prefetch accuracy in the last epoch is calculated and based on its value relative to two predetermined thresholds *Accuracy Thresh1* and *Accuracy Thresh2* we select the prefetch degree for the next epoch. If the accuracy is greater than *Accuracy Thresh2* we select a high prefetch degree; if it is below *Accuracy Thresh1* we select a low prefetch degree; and if it is in between the two thresholds we select a moderate prefetch degree.

E. What our Memory-Side Prefetching is NOT

To avoid any confusion, we would like to explicitly state that our proposal is:

- NOT a conventional memory-side prefetcher which have previously been proposed to push data pro-actively into caches or prefetch buffers on-chip. It is to be viewed more as an opportunistic transport of current row buffer data on-chip; this data is temporarily held in the buffer for anticipated need.
- NOT a normal Next-Line(s) Cache Prefetcher. While we do find spatial proximity within the row buffers prominent for opportunistically bringing in the data from the open row, we could have very well leveraged other access patterns within that row. Further, we will discuss later that our proposal does better than a normal Next-Line(s) Cache Prefetcher.
- NOT to be viewed as a replacement for core-side prefetching. It addresses some of the problems why one may not want to use a core-side prefetcher in a large-scale system, and does not preclude it. They can work in tandem and can be complementary, as our experimental evaluation clearly shows.
- NOT providing a shared Last Level Cache at the memory controllers. The proposed space is relatively small (1MB compared to 32MB of L2 cache in our default configuration), and even if it is exclusive, its capacity will hardly suffice as a cache. In fact, we later show that increasing the size of L2 by the size of the prefetch buffer and pushing prefetched data to it will not give as much benefits as our proposal.

IV. Hardware Implementation

Hardware implementation of our memory-side prefetching scheme involves two main parts. First, a prefetch buffer structure is added to each memory controller to store the prefetched data. Second, additional prefetch requests are issued on the memory channels and banks. In this section, we discuss the implementation issues and overheads of these two parts.

[3]The overheads of having an extra prefetch buffer in the memory controller are discussed in Section IV.

A. Prefetch Buffer Implementation

Our prefetch buffer is a small SRAM-based storage that keeps the data retrieved from memory by our memory-side prefetcher. This buffer is directly controlled by the memory controller and located together with the memory controller hardware. The prefetch buffer at each memory controller caters to different portions of memory and there is no need to maintain coherence between them. The organization of the prefetch buffer is shown in Figure 4. Although it is implemented as a single unit, the prefetch buffer, in fact, consists of a collection of per-core private prefetch buffers with additional resources. Each per-core prefetch buffer has k sets of w-ways, giving a total of $k \times w \times n$ entries, where n represents the number of cores in the system. Note that the prefetch buffer is implemented as a w-way set associative memory, and each prefetch buffer entry in one of w ways contains a single cache line. Therefore, for a prefetch degree greater than one, multiple entries are allocated in the prefetch buffer each time we prefetch new data.

The total *storage overhead* of the prefetch buffer can be calculated as follows. First, each prefetch buffer entry contains 64 bytes of data. For each prefetch entry, in addition to a valid bit, a tag field identifies the physical address bits (less the bits used to index into the target set) for the data in the entry. We assume a 32-core system with 128 prefetch buffer entries per core, 32-way associativity, and 40 bits of tag, in which case the total tag storage overhead is $32 \times 128 \times (40+1)/8 = 20.5$KB and the total data storage overhead is $32 \times 128 \times 64$B $= 256$KB, per prefetch buffer. Overall, with 4 prefetch buffers (one per memory controller), (256KB+20.5KB)$\times 4$ = 1.1MB additional storage is required, which is around 3% of the capacity of the 32MB L2 cache we used in the evaluation. We modeled the energy consumption of our prefetch buffer using CACTI [25] (assuming an SRAM with 65nm technology), which came out to be $0.87nJ$ for each read operation and $0.8nJ$ for each write operation, with a total area overhead of $2.85mm^2$ and a power overhead of less than 1W.

Each prefetch buffer has one write port and one read/write port. The write port receives the data prefetched from the memory and writes it into the array. The read/write port is used by the memory controller to check for prefetch hits and for the invalidation of prefetch entries in case of a memory write. Looking up the prefetch buffer requires an associative search across all entries in the target set. Writing prefetched data into the buffer requires invalidation of the least recently accessed prefetch buffer entry in the target set and overwriting that entry with the newly prefetched data. For memory writes (i.e., data sent from the core side to memory), we employ a *write-invalidate protocol*. When a write is received by our prefetch buffer, all prefetch buffer entries with the same address are invalidated. Hence, the tags of all entries in the target set must be checked, and in case a match occurs, those entries must be invalidated to ensure that stale data does not exist in the prefetch buffer. Note that the same data can exist in more than one core's prefetch buffers. Therefore, the invalidate command is broadcast to all private prefetch buffers inside the target memory controller. Note that the cache coherence protocol does not include the prefetch buffer, keeping it simple (non-coherent, like the DRAM itself). The only coherence we have to maintain is that between the prefetch buffer and the DRAM.

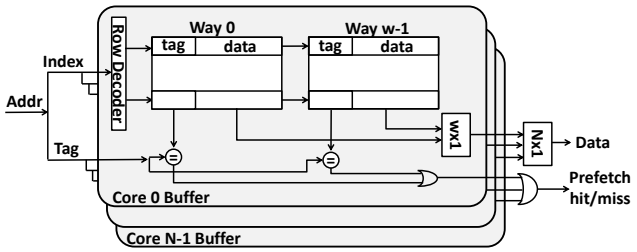

Fig. 4. Prefetch buffer organization.

B. Prefetch Request Timing

Our memory-side prefetch requests are generated internally by the memory controller and are scheduled contiguously right after a row buffer read hit occurs. The prefetch request keeps the memory channel busy for an additional $PD \times BL/2$ cycles, where BL is the *Burst Length*. Further, all memory requests that target the same bank and reside in the bank queue in the memory controller must wait for an additional $PD \times BL/2$ cycles due to the prefetch request.

V. Experimental Evaluation

A. Setup

We evaluate our memory prefetching scheme using the Simics [21] full-system simulator with GEMS [22]. We model a network-on-chip (NoC) based CMP with the MOESI_CMP_directory cache coherence protocol. We use Opal module of GEMS to simulate out-of-order cores. The cache and memory hierarchy are modeled using Ruby and the NoC is modeled using GARNET. Table IV shows the important processor and memory parameters used in our experiments, and their default values. Later, we conduct a sensitivity analysis on some of these parameters.

Processor	32 cores at 2.4 GHz; ultra-sparc-iii-plus ISA out-of-order; 12-stage pipeline; issue width: 4
Network On Chip	8x4 2D mesh network; 2-stage pipelined router, 4VCs per port with 4-entry buffer each; 128b flit
Caches	64-byte cache line; 32KB L1D; 32KB L1I per core; 32MB banked shared LLC (S-NUCA) L1 hit latency: 3 cycles; L2 hit latency: 6 cycles
Memory	16GB; DDR3-1600; 4 memory channels; total 51.2 GBps bandwidth 1 DIMM, 2 ranks, and 16 banks at each channel
Memory Parameters	Row Buffer Hit: 42 cycles; Conflict: 102 cycles t_{CL}, t_{RP} and t_{RCD} = 10, 10, 10 memory cycles
Address Mapping	Page Interleaving at the Memory Controllers Cache Line Interleaving at the L2 caches
Memory Prefetch Parameters	Hit latency: 5 cycles; Max degree: 4 lines Timing: Prefetch at Row Buffer Hit 256KB, 128-entry, 32-way set associative prefetch buffer at each memory controller
Memory Prefetch Opt. Thresholds	Accuracy Thresh1: 10; Accuracy Thresh2: 25 Queueing Thresh: 48 requests
Cache Prefetch Parameters	Max Degree: 4; Prefetch distance: 24 Prefetch Window: 32 cache lines Train & Stream entries per core: 64

TABLE IV. Configuration of the evaluation platform.

B. Benchmarks

We created multiprogrammed workloads from applications in the SPEC CPU2K6 and SPEC CPU2K benchmark suites and multi-threaded workloads from SPEC OMP2K1 applications

Application	L2 MPKI	RBHR	NLL	Application	L2 MPKI	RBHR	NLL	Application	L2 MPKI	RBHR	NLL
470.lbm	34.54	36.49	5.88	183.equake	7.12	72.16	68.21	465.tonto	0.61	70.90	43.43
462.libquantum	31.66	97.03	98.90	436.cactus	5.83	13.57	10.62	444.namd	0.59	77.64	56.50
459.GemsFDTD	30.47	37.29	43.17	429.mcf	5.08	34.83	2.12	464.h264ref	0.56	66.71	62.55
179.art	25.57	80.89	6.01	435.gromacs	4.57	57.31	62.46	416.gamess	0.47	57.55	44.36
433.milc	21.47	66.99	37.55	171.swim	4.33	17.51	18.54	481.wrf	0.43	71.49	36.85
401.bzip2	20.63	3.10	2.47	434.zeusmp	2.88	49.59	49.12	458.sjeng	0.42	18.47	15.97
437.leslie3d	18.36	72.86	72.54	450.soplex	2.24	20.13	15.52	403.gcc	0.39	69.13	54.64
410.bwaves	17.10	66.09	67.45	454.calculix	1.20	93.19	21.02	453.povray	0.26	79.63	54.87
483.xalancbmk	16.77	76.30	14.31	473.astar	1.12	35.55	19.94	447.dealII	0.21	78.08	59.41
471.omnetpp	12.95	48.30	3.74	456.hmmer	0.84	38.69	34.32	400.perlbench	0.15	74.76	51.51
482.sphinx3	11.99	29.50	26.25	445.gobmk	0.74	43.20	18.26				

TABLE V. MEMORY CHARACTERISTICS OF SPEC2006 APPLICATIONS.

to evaluate our memory-side prefetching scheme. For each application, first we did a study on its memory characteristics. We specifically looked at the MPKI, *Row Buffer Locality* and *Next Line Locality* of each application when running alone on a single core platform. Table V shows the MPKI, row buffer hit rate (RBHR) and next line locality (NLL) of all the considered applications. We created three workload categories (WL1 - High MPKI and High Locality, WL2 - High MPKI and Low Locality and WL3 - Low MPKI) with varying memory intensity and locality (NLL). We created a workload category (WL4) with high locality applications which benefit core-side prefetching and a workload category (WL5) with mix of all types of applications. We created two additional workload categories (WL6 - 8 applications with 4 threads each and WL7 - one application with 32 threads) with multithreaded applications. In each workload category, we created four workloads of 32 applications each by randomly selecting applications from the corresponding category.

While running a workload, each of the 32 applications is bound to an individual core. The simulations are fast forwarded to 15 billion cycles, the caches are warmed up for 5 million instructions and the detailed simulation is run for 10 million instructions on the first core. We measure the overall performance of a workload using the *harmonic mean of the IPCs* of individual applications in the workload which represents both fairness and performance [20]. We analyze the effect of prefetching on other metrics like on-chip and off-chip latencies, L2 hit rates and row buffer hit rates. We further measure the following prefetching-specific metrics:

$$PrefetchCoverage_{memory-side} = \frac{Prefetch\ hits}{Total\ memory\ requests} \times 100,$$

$$Prefetch\ Coverage_{core-side} = \frac{Prefetch\ hits}{Prefetch\ hits + misses} \times 100,$$

$$Prefetch\ Accuracy = \frac{Prefetch\ hits}{Number\ of\ prefetched\ lines} \times 100$$

Prefetch coverage is the percentage of memory requests served from the prefetch buffer out of the total requests to memory controller. In the case of core-side prefetcher, *prefetch coverage* is the percentage of L2 misses avoided by prefetched lines. *Prefetch accuracy* is a measure of how many prefetches are actually useful.

C. Results and Analysis

To evaluate the benefits of our memory-side prefetching approach (*MSP*), we compare it against four different prefetching schemes: a sophisticated stream-based core-side prefetching scheme (*CSP*), a next-line (degree 1) core-side prefetching scheme, our memory-side prefetching scheme which pushes the prefetched data to the on-chip (L2) caches (*MSP-PUSH*), an existing memory side prefetching work [17] proposed for unicore systems, extended to multicores (*IDLE-PUSH*). The core-side prefetcher (*CSP*) we implemented is an *adaptive-stride stream-prefetcher* similar to the one in [18] (and to the one adopted in current Intel Xeon [7] and IBM Power [31] processors) that builds streams based on the L2 misses and prefetches data into the last-level (L2) cache (which is increased by the size of the total prefetch buffer (1MB) in this case). We added accuracy-based "dynamic throttling" [31] to the core-side prefetcher to improve its effectiveness. In addition, we implemented techniques presented in [5] to reduce the effect of prefetch requests on demand requests at memory. We monitored the cache interference caused by our core-side prefetcher and identified that it is not a major problem in our configuration. We find that our core-side prefetcher has coverage and accuracy numbers similar to the prefetchers used in previous work [5], [16], [31]. We also implemented a simple next-line prefetcher at the core although it is not expected to give better benefits than the sophisticated stream-based core-side prefetcher. Our memory-side prefetching (*MSP*) scheme prefetches at row buffer hits and stores the data in prefetch buffers at the memory controllers (on-chip). (*MSP-PUSH*) scheme is similar to (*MSP*), but does not use the prefetch buffers and instead pushes the prefetched data to the on-chip (L2) cache which is increased by 1MB. (*IDLE-PUSH*) is similar to [17] which uses the idle periods on the memory bus to prefetch data from open rows and pushes the prefetched data to on-chip (L2) caches. In the combined case (*CSP+MSP*), both the prefetchers work independently where the memory-side prefetcher brings data to prefetch buffers from DRAM and core-side prefetcher fetches data from either the prefetch buffer or memory to the on-chip (L2) caches.

Figure 5 shows the percentage IPC improvements over the *no prefetching* case using five of the six schemes discussed above. The first bar for each workload shows the improvement with *CSP* scheme, the second, third, fourth and fifth bars show the improvements with the *MSP*, *MSP-PUSH*, *IDLE-PUSH*, and *CSP+MSP* schemes, respectively. The average IPC improvement over all 28 workloads (7 workload categories $\times$ 4 workloads/category) is 1.9% with the *CSP* scheme, -17.4% with next-line core-prefetcher (not shown in graph), 6.2% with *MSP*, 4.6% with *MSP-PUSH*, -3% with *IDLE-PUSH* and 10% with the *CSP+MSP* scheme. *CSP* scheme gave noticeable improvements (7.3%) only in workload category (WL4). Even by reserving a cache way just for prefetches the average IPC improvement with CSP increased to only 2.6%.

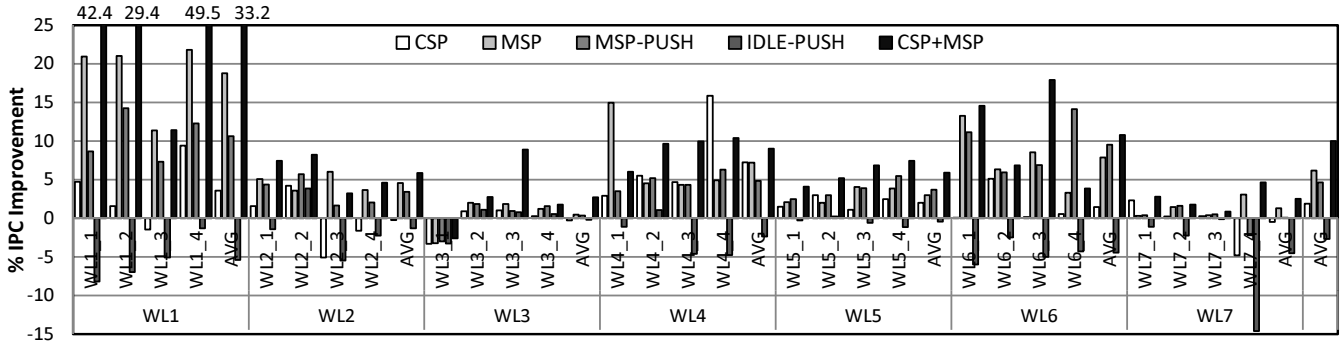

Fig. 5. Percentage IPC improvement over no prefetching with different prefetchers.

We investigate the reason for this poor performance of *CSP* in the following paragraphs. In almost all workloads, memory-side prefetcher (*MSP*) performed better than the core-side prefetcher (*CSP*). The difference between (*CSP*) and (*MSP*) is more noticeable in high MPKI workloads (WL1, WL2 and WL6). The *MSP-PUSH* scheme suffers from the on-chip resource contention and loses some performance compared to *MSP*. The *IDLE-PUSH* scheme was not able to bring any performance improvement on average because of the following problems: (i) useful idle periods on the memory bus are rare[4] as show in Figure 2, and (ii) the pushed data creates contention in the on-chip network and pollutes the caches. Accordingly, it performed well only in low MPKI workloads (WL3), and in all the other workloads its performance was worse than the baseline no prefetching case. The improvements obtained in the combined scheme (*CSP+MSP*) are higher than any other scheme in all the workloads. The improvements are especially good (average 33.2%) in high MPKI workloads (WL1). We want to emphasize that the improvement with *CSP+MSP* is larger than the sum of the improvements from *CSP* and *MSP* in many cases. This is because many core-side prefetches (in the combined case) hit in the prefetch buffer filled by the memory-side prefetcher in the *CSP+MSP* case which were originally accessing the DRAM and overwhelming the memory subsystem in the *CSP* case.

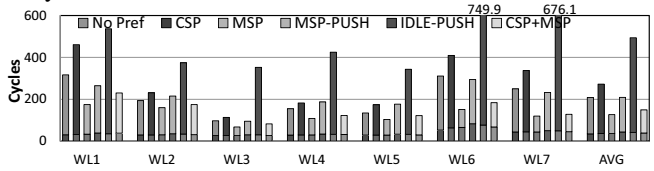

Fig. 6. Average on-chip and off-chip latencies for an LLC miss without prefetching and different prefetching schemes.

Figure 6 shows the average *on-chip* and *off-chip latencies* incurred by an L2 miss in different schemes (*No Pref*, *CSP*, *MSP*, *MSP-PUSH*, *IDLE-PUSH* and *CSP+MSP*). Six stacked bars are shown for each workload category. Each bar represents the average latencies in the four workloads in that category with a particular scheme. The first bar is the total miss latency without any prefetching, the second, third, fourth, fifth and sixth bars show the miss latencies with *CSP*, *MSP*, *MSP-PUSH*, *IDLE-PUSH* and *CSP+MSP* schemes respectively. Within each bar, the lower part shows the number of cycles spent by a request on-chip, and the upper part shows the number of cycles spent in the memory controller. It can be seen that the core-side prefetching, even with sophisticated techniques like throttling and prefetch aware memory controllers, is still causing significant delays at the memory controller (35.8% increase on average).[5] Our *MSP* scheme is very effective in reducing the off-chip latency (48.5% reduction on average) especially in high MPKI workloads (WL1, WL2, WL6 and WL7). *MSP-PUSH* and *IDLE-PUSH* schemes push all the prefetched data to caches and lead to higher on-chip latencies. They also show higher off-chip latencies because of the lack of prefetch buffer at memory controller in the case of *MSP-PUSH* and because of the interference of prefetch activity with demand requests in the *IDLE-PUSH* scheme. In the case of *CSP+MSP*, the increase in off-chip latency by the core-side prefetcher is balanced by the reduction in latency provided by the memory-side prefetcher.

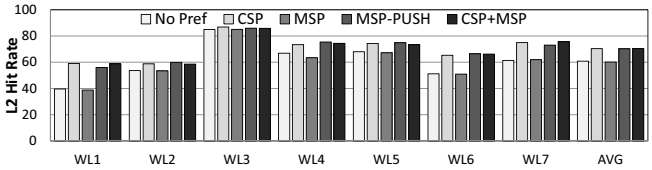

Fig. 7. L2 hit rates in different schemes.

Figure 7 plots the L2 hit rates without prefetching (*No Pref*) and with different prefetching schemes. The first bar for each workload category shows the average L2 hit rate in the no prefetching case and the next four bars show the average L2 hit rates in the *CSP*, *MSP*, *MSP-PUSH* and *CSP+MSP* schemes respectively. The average L2 hit rate over all workloads in the no prefetching case is 60.8% and the hit rates with *CSP*, *MSP*, *MSP-PUSH* and *CSP+MSP* are 70.3%, 60.1%, 70.2% and 70.4%. In all the schemes that prefetch to cache (*CSP*, *MSP-PUSH* and *CSP+MSP*) the L2 hit rate has improved significantly (around 19% on average). Memory-side prefetcher (*MSP*) prefetches to prefetch buffer and L2 hit rate is not affected.

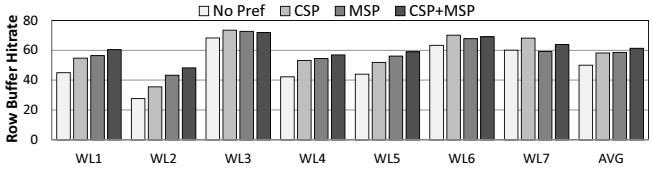

Fig. 8. Row buffer hit rates in different schemes.

Figure 8 plots the row buffer hit rates with different schemes.

[4]Note that this is not the case in a uniprocessor system where idle times are higher, and hence [17] demonstrated good improvements.

[5]We validated our implementation of CSP by repeating the experiments conducted in [5], [18], [31] for 4-core scenarios and observing similar results.

The first bar for each workload category shows the average row buffer hit rate in the no prefetching (*No Pref*) case and the next three bars show the average row buffer hit rates for the *CSP*, *MSP* and *CSP+MSP* schemes. The average row buffer hit rate over all workloads in *No Pref* case is 50.4% and the hit rates with *CSP*, *MSP* and *CSP+MSP* are 58.2%, 58.6% and 61.3%. In all the workloads, row buffer hit rate improved in the presence of core-side prefetching (22.2% on average), because of the locality of prefetch requests. Memory-side prefetching also increases the row buffer hit rate because of the memory prefetches which are essentially row buffer hits.

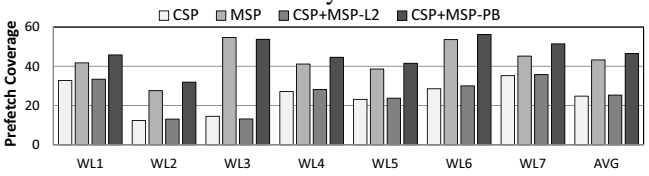

Fig. 9. Prefetch coverage in different prefetch schemes.

Figure 9 compares the *prefetch coverage* in different prefetch schemes. The first and second bars for each workload category show the average prefetch coverages in *CSP* and *MSP* schemes respectively. The third and fourth bars show the average prefetch coverage of *CSP+MSP* scheme at L2 and the average prefetch coverage of the *CSP+MSP* scheme at the prefetch buffer. It is important to distinguish between the prefetch coverage at L2 and prefetch buffer in the case of *CSP+MSP* scheme because of the difference in the savings provided by prefetch hits at these places. The average prefetch coverage over all workloads in the *CSP* case is 24.7%, and the coverages in *MSP* case is 43.2%. The average prefetch coverage with the *CSP+MSP* scheme is 25.3% at the L2 and 46.4% at the prefetch buffer. It can be seen that the coverage of memory-side prefetcher is greater than the coverage of core-side prefetcher in all the workload categories. This is because, the memory-side prefetcher incurs lesser overhead per prefetch than a core-side prefetcher and can afford to prefetch more data leading to more prefetch hits in the prefetch buffer.

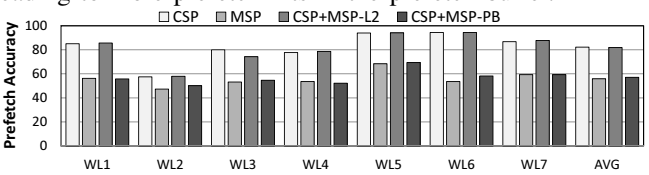

Fig. 10. Prefetch accuracy in different prefetch schemes.

Figure 10 compares the *prefetch accuracy* of the different prefetch schemes. The first bar for each workload category shows the average prefetch accuracy in *CSP* scheme, the second, third and fourth bars show the average prefetch accuracy in *MSP* scheme, *CSP+MSP* scheme at L2 and *CSP+MSP* scheme at the prefetch buffer. The average prefetch accuracy over all workloads in the *CSP* case is 82.2% and the accuracies in *MSP*, *CSP+MSP* at L2 and *CSP+MSP* at prefetch buffer are 55.9%, 81.8% and 57.1%. It can be seen that the accuracy of core-side prefetcher is consistently greater than the accuracy of memory-side prefetcher. This is due to the stream based core-side prefetcher's ability to monitor the complete miss pattern for an application and issue accurate prefetches while the memory side prefetcher is more opportunistic and prefetches along with row buffer hits leading to less accurate prefetches. Also, the percentage of prefetches out of the total memory requests is higher in the case of *MSP* (44.4%) than *CSP* (29.1%) leading to lesser accuracy.

Our memory-side prefetcher (*MSP*), being "DRAM state-aware" can also reduce the power consumption of DRAM by reducing the number of row activations and proactively retrieving data from open rows. In comparison to core-side prefetching, an inaccurate prefetch in our scheme (*MSP*) also wastes less energy as (i) it is done on an already-open row, and (ii) it does not use the on-chip network. The average DRAM power savings obtained with *MSP* (calculated using MICRON power calculator [24]) are 8.8% and 8.3% compared to no prefetching and *CSP* cases respectively.

From the above analysis it can be concluded that although core-side prefetcher (*CSP*) significantly improves L2 hit rates (18.9% on average as shown in figure 7) and does accurate prefetching (82.2% on average in figure 10), it is not very effective in improving the overall performance of the system (average 1.9% in figure 5). This is due to the increase in queueing latency at the memory controllers (35.8% on average as shown in figure 6) because of the bursty nature of core-side prefetch requests. To confirm this fact, we ran the same simulations with a constant memory latency (200 cycles) and the core-side prefetcher gave an average improvement of 10.4%. In fact, we could not find any work in recent literature which shows significant performance improvements with core-side prefetching on a many(32)-core system when full memory subsystem and on-chip network is modeled. We believe, this problem of off-chip queueing becomes more prominent as we move to larger core counts and core-side prefetching becomes far less effective. On the other hand, our memory-side prefetcher is a low overhead solution which can scale with the prefetching requirements of larger number of cores. However, pushing the prefetched data to on-chip caches is also not a good idea because of on-chip network and cache contention problems. Finally, we conclude that, combined prefetching (*CSP+MSP*) leverages the advantages of both the prefetching schemes and yields better benefits than either of them.

VI. Related Work

Hughes et al. [8] proposed the use of memory prefetching to improve the performance of programs working with linked data structures. They exploit the predictability of accesses available in linked data structures to do the prefetching. Their work focuses on a specific kind of programs whereas the idea presented in this paper in more versatile and works with any kind of programs. The work in [9] proposed an Adaptive Stream Detection (ASD) technique at the memory controller to identify the short-streams. They control the aggressiveness of the stream prefetcher using Stream Length Histogram (SLH) that are computed periodically. Their technique could stop useless prefetches. As indicated previously it is mainly a prediction technique which dictates *what to prefetch?* while ours is a complete memory prefetching solution which can actually incorporate their prediction technique. Impulse [2] proposed the use of a smarter memory controller which does address remapping and prefetching for better performance. Both these schemes do not take into account the state of the channel or the row buffers before prefetching. In contrast, the

work presented in this paper considers the current memory state before issuing the prefetches so as to minimize the overhead of prefetching.

The work presented in [17] proposed the use of prefetchers both in the L2 caches and the memory controllers to improve performance. They identify idle cycles on the memory channel and utilize them to schedule the prefetch requests. The work in [27] proposed a hybrid software/hardware prefetch scheme. In their scheme, hardware prefetcher does prefetching into the L1 cache, while the compiler-directed special load and prefetch instructions bypass the data into the registers directly. The work presented in [12] uses Markov Predictors for memory-side prefetching in a uniprocessor system. They also proposed the use of on-chip prefetch buffers along with L1 caches. [30] proposed a user-level memory thread based software prefetching. This user-level memory thread runs on a general purpose core present on the memory controller chip. All these techniques are unicore based and employ a push-based strategy where the prefetched data is pushed to the on-chip entities which leads to network congestion and cache pollution in a large-scale on-chip network based CMP.

VII. Concluding Remarks

This paper has re-visited memory-side prefetching in the context of large-scale CMPs, leveraging two increasingly important observations: (i) high off-chip latencies due to limited memory channel bandwidths and row-buffer conflicts of DRAMs make it imperative to avail of opportunistic and instantaneous knowledge of main memory to bring data on-chip ahead of its need; and (ii) on-chip resource contention (in the caches and network) can increase with ill-timed and inaccurate prefetches. By simply bringing the data into an on-chip prefetch buffer maintained at the memory controllers, a *midway* solution offered by our memory-side prefetcher cuts on average 48.5% of the latency of an L2 read miss that would otherwise go to the DRAMs. Our solution does better than existing core-side prefetchers or memory-side prefetchers which push data to caches in a large-scale CMP (like 32 cores), where their effectiveness decreases. Further, it can work in tandem with core-side prefetcher to amplify the benefits. Using a spectrum of multiprogrammed and multithreaded workloads, we have shown that this memory-side prefetcher provides IPC improvements of 6.2% (maximum of 33.6%), and 10% (maximum of 49.6%), on an average when running alone and when combined with a core-side prefetcher, respectively. We believe that this work introduces several possibilities for future extensions, where the two prefetchers can be further individually optimized to become aware of the other, with a rich set of options for locating the *midway* meeting point.

References

[1] M. Cade and A. Qasem, "Balancing locality and parallelism on shared-cache multicore systems," in *HPCC*, 2009.

[2] J. Carter *et al.*, "Impulse: Building a smarter memory controller," in *HPCA*, 1999.

[3] T.-F. Chen and J.-L. Baer, "A performance study of software and hardware data prefetching schemes," in *ISCA*, 1994.

[4] B. T. Davis, "Modern DRAM architectures," Ph.D. dissertation, 2001.

[5] E. Ebrahimi *et al.*, "Prefetch-aware shared resource management for multicore systems," in *ISCA*, 2011.

[6] E. Ebrahimi *et al.*, "Coordinated control of multiple prefetchers in multicore systems," in *MICRO*, 2009.

[7] R. Hegde, "Optimizing application performance on intel core microarchitecture using hardware-implemented prefetchers," *Intel*, 2008.

[8] C. J. Hughes and S. V. Adve, "Memory-side prefetching for linked data structures for processor-in-memory systems," *Journal of PDC*, 2005.

[9] I. Hur and C. Lin, "Memory prefetching using adaptive stream detection," in *MICRO*, 2006.

[10] S. Iacobovici *et al.*, "Effective stream-based and execution-based data prefetching," in *ICS*, 2004.

[11] B. Jacob *et al.*, *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2007.

[12] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in *ISCA*, 1997.

[13] M. Karlsson *et al.*, "A prefetching technique for irregular accesses to linked data structures," in *HPCA*, 2000.

[14] C. Kim *et al.*, "Nonuniform cache architectures for wire-delay dominated on-chip caches," *Micro, IEEE*, vol. 23, no. 6, nov.-dec. 2003.

[15] Y. Kim *et al.*, "Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA*, 2010.

[16] C. J. Lee *et al.*, "Prefetch-aware DRAM controllers," in *MICRO*, 2008.

[17] W.-f. Lin, "Reducing DRAM latencies with an integrated memory hierarchy design," in *HPCA*, 2001.

[18] G. Liu *et al.*, "Enhancements for accurate and timely streaming prefetcher," *The Journal of ILP*, vol. 13, Jan. 2011.

[19] C.-K. Luk *et al.*, "Profile-guided post-link stride prefetching," in *ICS*, 2002.

[20] K. Luo *et al.*, "Balancing thoughput and fairness in smt processors," in *ISPASS*, 2001.

[21] P. S. Magnusson *et al.*, "SIMICS: A full system simulation platform," *Computer*, vol. 35, no. 2, Feb. 2002.

[22] M. M. Martin *et al.*, "Multifacets general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, 2005.

[23] Micron, "DataSheet: 1Gb DDR3 SDRAM."

[24] Micron, "DDR3 Power Calculator."

[25] N. Muralimanohar *et al.*, "Optimizing nuca organizations and wiring alternatives for large caches with CACTI 6.0," in *MICRO*, 2007.

[26] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *ISCA*, 2008.

[27] D. Ortega *et al.*, "Cost-effective compiler directed memory prefetching and bypassing," in *PACT*, 2002.

[28] D. K. Poulsen and P.-C. Yew, "Data prefetching and data forwarding in shared memory multiprocessors," in *ICPP*, 1994.

[29] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, no. 12, Dec. 1978.

[30] Y. Solihin *et al.*, "Correlation prefetching with a user-level memory thread," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 6, Jun. 2003.

[31] S. Srinath *et al.*, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *HPCA'07*.

[32] K. Sudan *et al.*, "Micro-pages: increasing dram efficiency with locality-aware data placement," in *ASPLOS*, 2010.

[33] C.-J. Wu *et al.*, "Pacman: prefetch-aware cache management for high performance caching," in *MICRO*, 2011.

[34] Y. Wu, "Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching," in *PLDI*, 2002.

[35] C.-L. Yang and A. R. Lebeck, "Push vs. pull: data movement for linked data structures," in *ICS*, 2000.

[36] Z. Zhang *et al.*, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *MICRO*, 2000.

[37] X. Zhuang and H.-H. S. Lee, "A hardware-based cache pollution filtering mechanism for aggressive prefetches," in *ICPP*, 2003.