Bundling Spatially Correlated Prefetches for Improved Main Memory Row Buffer Locality

by

Patrick Judd

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright by Patrick Judd 2014
Bundling Spatially Correlated Prefetches for Improved Main Memory Row Buffer Locality

Patrick Judd

Masters of Applied Science

Graduate Department of Electrical and Computer Engineering
University of Toronto

2014

Abstract

Row buffer locality is a consequence of programs’ inherent spatial locality that the memory system can easily exploit for significant performance gains and power savings. However, as the number of cores on a chip increases, request streams become interleaved more frequently and row buffer locality is lost. Prefetching can help mitigate this effect, but more spatial locality remains to be recovered. In this thesis we propose Prefetch Bundling, a scheme which tags spatially correlated prefetches with information to allow the memory controller to prevent prefetches from becoming interleaved.

We evaluate this scheme with a simple scheduling policy and show that it improves the row hit rate by 11%. Unfortunately, the simplicity of the scheduling policy does not translate these row hits to improved performance or power savings. However, future work may build on this framework and develop more balanced policies that leverage the row locality of Prefetch Bundling.
Acknowledgments

There are many people to thank as I reflect back on my Master’s degree. First and foremost I must thank my supervisor, Andreas Moshovos, for his support, confidence in my abilities, and most of all for his ability to keep me grounded during stressful and trying times. I would also like to thank the members of the AENAO research group for their help, support and genuine camaraderie. Special thanks go to Jason Zebchuk for his help and guidance in learning and wrestling with Flexus. I would also like to thank Myrto Papadopoulou for her technical and moral support, and for tolerating my uncanny ability to kill cluster machines.

I would also like to thank Andreas Moshovos, the University of Toronto and the Natural Sciences and Engineering Research Council of Canada for their generous financial support.
# Table of Contents

1 Introduction ........................................................................................................................................... 1
   1.1 Contributions ................................................................................................................................. 4
   1.2 Thesis Structure ............................................................................................................................... 5

2 Background ........................................................................................................................................... 6
   2.1 Spatial Memory Streaming .............................................................................................................. 6
      2.1.1 Overview ................................................................................................................................. 6
      2.1.2 Hardware Design ....................................................................................................................... 7
   2.2 Dynamic Random Access Memory ................................................................................................. 9
      2.2.1 Memory Geometry .................................................................................................................... 10
      2.2.2 DRAM Commands .................................................................................................................... 12
      2.2.3 Row Buffer Management Policies .......................................................................................... 12
      2.2.4 Memory Scheduling .................................................................................................................. 13
      2.2.5 Address Mapping ....................................................................................................................... 14

3 Motivation ............................................................................................................................................ 15
   3.1 Trace Based Simulation ................................................................................................................... 15
   3.2 Timing Simulation ........................................................................................................................... 17

4 Design ................................................................................................................................................... 19
   4.1 Single Packet Bundling vs. Tagged Bundling ............................................................................... 20
      4.1.1 Single Packet Bundle ............................................................................................................... 20
      4.1.2 Tagged Bundle ......................................................................................................................... 21
   4.2 Prefetch Bundling Design .............................................................................................................. 24
   4.3 Prefetch Bundling Implementation ............................................................................................... 25
      4.3.1 Prefetch Controller ................................................................................................................... 25
      4.3.2 Caches and Interconnect .......................................................................................................... 26
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2 Adaptive Stream Detection</td>
<td>58</td>
</tr>
<tr>
<td>7.3 Scheduled Region Prefetching</td>
<td>58</td>
</tr>
<tr>
<td>7.4 Prefetch-Aware DRAM Controllers</td>
<td>59</td>
</tr>
<tr>
<td>7.5 Footprint Cache</td>
<td>59</td>
</tr>
<tr>
<td>7.6 Complexity Effective Memory Access Scheduling</td>
<td>60</td>
</tr>
<tr>
<td>8 Conclusion</td>
<td>61</td>
</tr>
<tr>
<td>8.1 Future work</td>
<td>62</td>
</tr>
</tbody>
</table>
List of Tables

Table 1. SMS and system parameters ............................................................................. 29
Table 2. Prefetch Bundling parameters ............................................................................ 29
Table 3. Optimized Prefetch Bundling parameters ......................................................... 31
Table 4. CMP parameters ................................................................................................. 34
Table 5. SMS parameters ................................................................................................. 34
Table 6. DRAM parameters .............................................................................................. 35
Table 7. Workloads ........................................................................................................... 36
List of Figures

Figure 1. Access stream interleaving .......................................................................................... 2
Figure 2. Example of the Active Generation Table learning a spatial pattern ......................... 8
Figure 3. PHT and prefetch generation with PC/offset indexing .................................................. 9
Figure 4. Logical memory organization. 1 channel, 2 ranks, 4 banks per rank ....................... 11
Figure 5. Address mapping ........................................................................................................ 14
Figure 6. Row hit rate for single core and multi-core systems for various queue sizes ............ 15
Figure 7. Timing results comparing the actual and projected row hit rate for a 16 core system with a 32 entry queue ........................................................................................................ 17
Figure 8. Timing diagram for a) Single Packet Bundles and b) Tagged Bundles ....................... 23
Figure 9. Modified SMS backend ............................................................................................... 26
Figure 10. Modified memory controller with Prefetch Bundle Queue ....................................... 27
Figure 11. Prefetch bundle size distribution. The leftmost bar is 2-prefetch bundles, the rightmost bar is 31-prefetch bundles ........................................................................................................ 40
Figure 12. Prefetch Fill Level Distribution ................................................................................... 41
Figure 13. Prefetches as a percentage of total memory requests ................................................ 43
Figure 14. Memory Transactions per kilo-instruction .................................................................. 44
Figure 15. Row hit rate for a 2 memory controller configuration ................................................ 46
Figure 16. Row hit rate versus the number of memory controllers ............................................. 47
Figure 17. Average request latency. 2 memory controllers ......................................................... 48
Figure 18. Memory requests per 1000 cycles. 2 memory controllers ........................................ 49
Figure 19. Average prefetch and demand request latency. 2 memory controllers .................... 50
Figure 20. Bandwidth per channel. 2 memory controllers ................................................. 51
Figure 21. Memory power consumption [W]. 2 memory controllers ..................................... 52
Figure 22. Memory power breakdown [W]. 2 memory controllers ........................................ 53
Figure 23. Relative UIPC for 1, 2, and 4 memory controller systems ...................................... 55
Figure 24. L1D Miss rate normalized to the baseline. 4 memory controllers ......................... 56
1 Introduction

Dynamic random access memory (DRAM) based main memory is a primary bottleneck in modern computer systems, both in terms of performance and power. Over the past 20 years DRAM has failed to keep up with the rapid performance scaling of the central processing unit (CPU). Between 1990 and 2004 the CPU clock frequency has increased by a factor of 150 while DRAM latency has only improved by a factor of 4 [27].

DRAM has also become a major power consumer in servers. In 2007, Google reported DRAM contributing 30% of the hardware power consumption in one of their data centers [5]. And the relative power consumption of DRAM appears to be increasing. IBM reported DRAM contributing 28% of the total system power consumption for Power6 (2007) based servers and 46% for Power7 (2010) based servers [25]. This is the result of improvements in power saving techniques in the Power7 processor design and the increased memory capacity demanded by the growing number of cores.

One of the ways that modern systems mitigate the DRAM bottleneck is by exploiting row buffer locality. Row buffer locality describes memory accesses that access the same row in DRAM. While externally a memory reference typically transfers 32B-128B, data internally in the DRAM is accessed from a DRAM array in the granularity of much larger (4kB-16kB) rows, which is stored in intermediate storage called the row buffer. Subsequent accesses to the same row can read from or write to the row buffer without accessing the array. These accesses are called row hits. This can be exploited to the benefit of both performance and power. Accesses to the row buffer have lower latency than rows located in the DRAM array. In fact, DRAM array access incurs 3x the latency of accesses to the row buffer [3]. DRAM array accesses also consume a significant portion of overall DRAM power. This is due to the significant amount of current needed to charge and discharge the cell capacitors and bit lines. As such, increasing the number of row buffer accesses can have a significant impact on performance and power.
Program threads naturally exhibit a high amount of spatial locality, and thus, a high amount of row buffer locality. However, with the increasing number of cores in modern chip multiprocessors (CMPs), these spatially correlated access streams from each core get interleaved on their way to memory and much of that row buffer locality is lost [20]. For example, consider an application adding a constant to each element of an array as shown in Figure 1 a). When run on single core, the application naturally exhibits row locality as subsequent accesses touch consecutive memory addresses. When this application is parallelized however, this natural spatial locality may be lost. The array may be conceptually partitioned into continuous chunks which are then processed in parallel as shown in Figure 1 b). While the individual threads maintain the inherent locality of the original application, accesses from different threads proceed concurrently through the memory hierarchy and arrive tangled at the memory controller. As a result, the inherent spatial locality of the application may be lost. Careful data partitioning may reduce these effects but execution time differences across threads and cores would still perturb the access arrival order and thus the observed row locality.

**Figure 1. Access stream interleaving**
Many techniques have been proposed to exploit row buffer locality and increase the number of row hits. Among them are memory scheduling policies where requests are buffered just before the DRAM and are scheduled according to some policy. While requests arrive in some order, the order in which they are presented to main memory may be different. As a result, two accesses to the same row that are separated by other accesses may be scheduled back-to-back resulting in a row hit. For example, First Ready First Come First Served is a simple memory scheduling policy designed to exploit all the row buffer locality available in the memory controllers queues and has been shown to be effective at improving DRAM performance [17]. Memory scheduling policies like this one can mitigate the effects of interleaving, but they are limited by the size of the request queue and rate at which requests arrive. In Chapter 3 we demonstrate that First Ready First Come First Served (FR-FCFS) is not able to recapture all the lost spatial locality of a single thread in a multi-core system.

Other techniques to improve row buffer locality involve remapping blocks in memory to the same row [20] and memory side prefetching [6]. These are discussed in more detail in Chapter 7.

Spatially correlated prefetching can also reduce the amount of interleaving by issuing prefetches in batches that arrive at memory close together in time. The Spatial Memory Streaming (SMS) prefetcher is particularly attractive because it issues prefetches in batches that will always access the same row. We also demonstrate that SMS improves the overall row hit rate of the system. However, prefetches that are issued together in a batch experience delays as they travel through the cache hierarchy which causes them to spread apart in time. As a result, these prefetch batches still experience interleaving at the memory controller and potential row hits are lost, as we show in Chapter 3.

We propose a scheme called Prefetch Bundling, which bundles spatially correlated prefetches such that the memory controller can recover the row locality of interleaved prefetch streams. In this way, all of the potential row hits in each bundle are realized and the overall row hit rate is increased. We do this by tagging each prefetch with information about other prefetches in the bundle. The memory controller is augmented with a specialized bundle queue to buffer these prefetches and wait for the entire bundle to arrive before issuing to memory. In this way, Prefetch Bundling is able to offer more row buffer locality than FR-FCFS by looking beyond the
contents of the memory controller queues and consider pending prefetches when making scheduling decisions.

Using detailed timing simulations of a 16 core CMP and DRAM, we evaluate a simple scheduling policy that leverages Prefetch Bundles to maximize the row hit rate. We achieve an 11% increase in the row hit rate. Unfortunately, with this simple scheduling policy we do not see any appreciable speedup or power savings relative to our baseline system with SMS prefetching. However, the prefetch bundle queue offers the flexibility to explore the trade-off between latency and row hit rate and allows for future work on more sophisticated scheduling policies that balance row buffer locality, fairness and parallelism.

1.1 Contributions

This thesis makes the following contributions:

- We demonstrate that row locality decreases with the increasing number of cores in a shared memory CMP. Specifically we show that the row hit rate drops from 67% in a single core system to 33% in a 16 core system. We also show that SMS prefetching on its own significantly improves the row hit rate.

- We propose a scheme to bundle spatially correlated prefetches together so that their locality can be fully exploited to increase the row buffer hit rate. This scheme is also flexible enough to work with a variety scheduling policies.

- We propose a scheduling policy to maximize row hits with our bundling scheme.

- We evaluate our bundling scheme and scheduling policy and show that it can increase the row hit rate by 11%, though this does not result in any appreciable performance improvement or power savings.
1.2 Thesis Structure

The rest of the thesis is organized as follows. Chapter 2 provides the necessary background on SMS prefetching and DRAM operations. Chapter 3 describes our motivation and presents our motivating results. Chapter 4 describes the design of our Prefetch Bundling scheme and scheduling policy. Chapter 5 describes our simulation methodology. Chapter 6 presents a characterization of our workloads and an evaluation of design. Finally Chapter 7 offers our concluding thoughts and a description of potential future work.
2 Background

In this chapter we provide background on the two key components of our design, the SMS prefetcher and the DRAM memory system. In Section 2.1 we provide an overview of SMS. Our design relies on the fact that SMS generates multiple prefetches to the same aligned region of memory and issues these prefetches consecutively. It is also important to understand the back end hardware involved in generating prefetches since we modify it in our design. In Section 2.2 we describe the structure and fundamental characteristics of DRAM. Most importantly we describe why row buffer hits save power and improve performance.

2.1 Spatial Memory Streaming

Spatial Memory Streaming (SMS)[19] is a prefetcher that is able capture complex, repetitive data access patterns that span several kilobytes of data. SMS exploits a strong correlation between program instructions and access patterns and is able to accurately predict complex patterns of accesses to previously unseen data. It is also able to differentiate between multiple independent access patterns that are observed simultaneously in the program's execution.

2.1.1 Overview

SMS works by dividing the memory space into fixed sized spatial regions which are comprised of a contiguous set of cache blocks. Access patterns are recorded using bit vectors called spatial patterns, where each bit in the vector corresponds to a cache block in a spatial region. The size of this bit vector is equal to the number of cache blocks in a spatial region. A bit in the spatial pattern is set to 1 if that block is accessed as part of the pattern. The index of the bit in the spatial pattern corresponds to the offset of the cache block within the spatial region, called the spatial region offset. The spatial region is typically chosen as a power of two and aligned in memory, so the spatial region offset can be extracted directly from the less significant bits of the address. The higher order bits then represent the spatial region address.
SMS learns access patterns by observing accesses to a spatial region over a limited period of time, called the spatial region generation. The first access to the spatial region during a generation is called the trigger access. The end of a spatial region generation is determined by the first eviction or invalidation of a cache block within the spatial region. This serves as an appropriate limit to the spatial region generation because we do not want prefetched blocks to be evicted before they are accessed.

Spatial patterns are associated with both the program counter (PC) and the spatial offset of their trigger access. For repetitive access patterns we expect the same instructions to access data with the same relative offsets from each other, but not necessarily the same data those instructions have seen before. To predict these accesses, SMS stores the observed spatial patterns in a table indexed by the trigger accesses PC. When SMS observes an access with the same PC and offset it issues prefetches for all the blocks, excluding the trigger access, in the associated spatial pattern. In this thesis we refer to the set of prefetches generated by the spatial pattern as a batch.

2.1.2 Hardware Design

SMS is made up of two hardware structures, the active generation table (AGT), which observes the access patterns of currently active generations, and the pattern history table (PHT), which stores previously observed patterns for future predictions.

The AGT is a set-associative cache-like structure. Entries are tagged with the higher order bits of the address and contain the PC and spatial region offset of the trigger access and the spatial pattern bit vector. When a trigger access occurs a new entry is allocated in the AGT, setting the tag, PC and offset, initializing the spatial pattern to all zeros, and setting the bit corresponding to the trigger access to 1. On a subsequent access, the corresponding bit in the spatial pattern is set to 1. At the end of the spatial region generation, the entry is evicted from the AGT and the PC, offset and spatial pattern are sent to the PHT.
Figure 2 shows an example of the AGT learning a spatial pattern. In this example the program accesses spatial region A with reads to blocks at offsets of 1,0 and 3 within the spatial region.

The AGT is implemented as two tables, the accumulation table and the filter table. The filter table only stores a tag, PC and offset for each entry and is used to filter out spatial regions with only one access. Initially the trigger access allocates in the filter table and if a second access to that spatial region is observed then the entry is transferred to the accumulation table.

The PHT is a set associative cache-like structure. Entries are tagged by the PC and spatial region offset and contain the corresponding spatial pattern that was observed by the AGT. On a trigger access, SMS performs a lookup in the PHT. If a matching PC and offset are found, then prefetches are generated using the region base address of the new trigger access and the spatial pattern stored in the PHT. The bit corresponding to the trigger access is ignored since that cache block is already being request by the trigger access. An example of a PHT lookup and prefetch generation is given in Figure 3.

An optimization to the PHT is to use rotated patterns. This scheme rotates the spatial pattern so that the trigger offset is the first bit before storing in the PHT. In this way each entry is indexed only by the PC and one pattern will work for any trigger access offset.
The stream of generated prefetches are filtered to remove duplicates before being issued to the L1 cache. New prefetches are filtered against the set of blocks that exist in the L1 cache and the set of prefetches that are currently in flight. SMS keeps a separate record of the blocks that exist in the cache so that it can perform this filtering without causing an excessive number of tag lookups in the L1 cache for each potential prefetch.

2.2 Dynamic Random Access Memory

In Dynamic Random Access Memory (DRAM), bits are stored as the charge of a capacitor where a charge capacitor holds a 1 and a discharged capacitor holds a 0 [7]. A DRAM cell consists of one capacitor to hold the charge and one transistor to allow for charging and discharging. The transistor is controlled by the word line and connects the capacitor to bit line. To read from a cell, the bit line is precharged to half of the supply voltage and the word line is raised allowing charge to either flow into or out of the cell. The bit line is connected to a sense amplifier that will detect whether the voltage on the bit line goes up or down, amplify this change in voltage to a full logic 0 or 1 and hold the value in the sense amplifier latch. To write to the cell the value stored in the latch is driven on the bit line and the word line is raised so that current can flow in to, or out of, the capacitor. The reading process is destructive in that it
ultimately drains the charge from the capacitor, so the data in the sense amplifier latch must eventually be written back in to the cell. The capacitor in the cell is also imperfect and will leak charge over time. As such, an idle cell must be periodically refreshed by reading out the value and writing it back.

2.2.1 Memory Geometry

DRAM cells are physically organized in a 2-dimensional array of rows and columns. Cells in the same row are connected together by the word line, meaning that all cells in a row are either read or written at the same time. Cells in the same column are connected to a single sense amplifier by the bit line, so only one cell in a column can be accessed at a time. To perform either a read or write to the array, the entire corresponding row must first be read in to the sense amplifier latches, collectively called the row buffer. This is called opening the row. Data is typically accessed from the row buffer 4 to 16 bits at a time. The entire row must be written back in to the array before another row can be accessed. This is called closing the row. The row decoder selects the word line corresponding to the row address and the column decoder selects the bits from the row buffer that are being accessed.

A single DRAM integrated circuit, called a device, typically contains multiple DRAM arrays, called banks, each with their own row buffer. As such, transferring data between the row buffer and the array can be done simultaneously for each bank. However, DRAM devices have a small number of pins for data transfer. This number is called the device width and is equal to the width of column data accesses at the row buffer. Only one bank can connect to the data pins at a time.

Multiple devices are connected together to form a rank. The devices are connected in parallel to create a wider memory. The devices in a rank operate in lock step, meaning the same bank, row and column are accessed on each device for every operation. This results in a wider rows and wider accesses to data in the combined row buffer. These wider rows are also called DRAM pages. The device width and number of devices are chosen such that the total data pin width matches the width of the bus, typically 64 bits, that connects the memory to the processor. Multiple ranks connect to the same data bus so only one rank can use the data bus at one time.
Multiple ranks can also be divided into channels. Each channel is comprised of its own 64-bit data bus and the accompanying control signals used to select the desired rank, bank, row and column. There are different ways in which multiple channels can be used. For example, two 64-bit channels may be combined to form a single logical 128-bit channel, and the selected ranks within each channel operate in lock step to deliver 128 bits at a time. Conversely, two 64-bit channels may be treated as two separate partitions of ranks, to be operated independently. Independent channels may be controlled by a single memory controller, or each channel may have its own memory controller. In many-core systems it is beneficial to have multiple memory controllers distributed around the multiprocessor die to distribute the load of memory requests on the interconnect network.

Figure 4. Logical memory organization. 1 channel, 2 ranks, 4 banks per rank
2.2.2 DRAM Commands

Commands are sent over the command bus from the memory controller to the memory modules. There are five basic commands that, issued with an accompanying address, govern the operation of the memory modules: activate, column read, column write, precharge and refresh. Activate reads a row from a DRAM array into the row buffer. Column read and write commands transfer data columns from and to the row buffer. One column read or write command results in a burst of data transfer on the bus. Typically, for a 64 Byte cache block and a 64 bit data bus, an 8 cycle burst is required where 8 consecutive columns of data are transferred one cycle at a time. Precharge writes the data from the row buffer back into the array and then precharges the bit lines in preparation for the next Activate.

Refresh transfers the data from a row to the row buffer and then writes it back to the array. Unlike the other commands, refreshes are typically done on the same row, or rows, across all banks in a rank. One refresh command only refreshes a few rows at a time and the DRAM device keeps track of the last refreshed row.

2.2.3 Row Buffer Management Policies

Transferring data between the row buffer and the array and transferring the data between the row buffer and the memory controller often take the same number of clock cycles. Accessing the array also consumes a significant amount of energy in charging or discharging the cell capacitors and bit lines for an entire row. As such, the management policy of the row buffer will significantly affect the average request latency and power consumption. There are two basic policies on how long a row of data, or page, should be kept in the row buffer. The open page policy says to keep the page open in the row buffer until another row is accessed. The close page policy says to close the page immediately after use.

The open page policy is beneficial when DRAM accesses exhibit high row buffer locality. Row buffer locality refers to the reuse of data in the row buffer for multiple column
accesses. Keeping the row in the row buffer allows other access to perform a column access without first sending an activate command. In essence, the row buffer acts as a single entry cache for the DRAM bank, holding recently used data to reduce access latency. As such, an access to the row in the buffer is called a \textit{row hit} and an access to another row is called a \textit{row miss} or a \textit{row conflict}.

The close page policy is beneficial when there is little row buffer locality. For each access the memory controller must only send an activate command before accessing the data from the row buffer. With open page, a conflicting row request must send a precharge command and an activate command before accessing the row buffer. The closed page policy has lower latency for row conflicts because it performs the precharge ahead of the arrival of the conflicting request.

2.2.4 Memory Scheduling

Memory scheduling is the process of deciding the order in which to issue commands to memory. This can be broken into two phases, transaction scheduling and command scheduling. Transactions are the memory requests that are received from the processor, i.e. reads and writes. Commands are the individual steps to perform a transaction with DRAM. The basic DRAM commands are described in Section 2.2.2. Incoming transactions are placed in a transaction queue. The transaction scheduler chooses transactions, converts them to commands and forwards them to one of the command queues. The memory controller typically contains multiple command queues, such as one command queue per rank or one command queue per bank. The command scheduler then decides which command to issue based on the contents of the command queue and the state of the DRAM.

First Ready First Come First Served (FR-FCFS) is a simple scheduling policy that attempts to maximize row hits. Transactions are processed in order as long as there is room in the command queue. In the command queue, FR-FCFS prioritizes row hits over other requests. FR-FCFS will only close a row if there are no row hits available.
2.2.5 Address Mapping

The address mapping determines the channel, rank, bank, row and column for a given address. The choice of address mapping scheme has a direct impact on memory system performance. Since programs exhibit spatial locality, the lower order bits of the requested address will change more frequently than the higher order bits. Consider a stream of sequential block accesses to memory. If the lowest order bits select the column then sequential accesses will primarily access the same row and experience high row buffer locality. If the lowest order bits select the channel, rank or bank then sequential accesses will be serviced in parallel and contribute to higher bandwidth. Independent channels offer the most parallelism since they use separate physical buses and potentially separate memory controllers. Ranks and banks on the other hand only offer parallelism in array accesses, but must serialize the transfer of data over the shared bus. Also, switching ranks incurs more delay than switching banks. Rows are generally mapped to the highest order bits to minimize the frequency of row buffer conflicts.

Address mapping schemes are commonly denoted by listing the mapping of dimensions from most significant to least significant bit. For example, the mapping shown in Figure 5 is denoted as row-rank-bank-channel-column.

![Address mapping](image)

**Figure 5. Address mapping.**
3 Motivation

This chapter presents our motivating results which examine the effects of interleaving memory request streams on row buffer locality. In Section 3.1 we present our initial trace based simulation results. In Section 3.2 we present a more detailed analysis using timing simulations.

3.1 Trace Based Simulation

To motivate this work we look at the row hit rate of main memory for our workloads. We want to see how row locality is affected as we go from a single core system to a 16 core system. We also want to see how the inherent locality of SMS prefetches improves row locality and whether there is room for further improvement.

![Figure 6. Row hit rate for single core and multi-core systems for various queue sizes](image)
To generate our initial motivating results we use a functional simulation and trace analysis. This methodology is described in more detail in Section 5.3. In functional simulation loads and stores propagate through the cache to memory and return a reply in a single cycle. In this way we are only able to generate traces that capture the order of requests as they arrive at memory but not the timing of their arrival.

In Figure 6 we show the row hit rate averaged over our set of workloads. First we look at the effect increasing the number of processor cores has on the row hit rate experienced by memory by comparing the 1cpu and 16cpu bars in Figure 6. A single core system is able to expose the spatial locality of a single thread directly to the memory controller while in a 16 core system the request streams from each processor would be interleaved and that intra-thread locality would be perturbed. This can be seen in the results for the single entry queue where increasing the number of cores from 1 to 16 results in a drop in the row hit rate from 47% to 12%. In a system with a realistically sized 32 entry queue the row hit rate increases to 67% for a single core and 33% for 16 cores. In both cases, increasing the number of cores from 1 to 16 reduces the row hit rate by 35% of the total requests.

Next we add SMS prefetching to both our single core and multi-core system. SMS predicts requests that will all map to the same row in DRAM and issues the prefetches in a stream. In essence it is increasing the temporal locality of requests with high spatial locality. By issuing these requests closer together in time, SMS reduces the chance that other requests will become interleaved and cause row conflicts.

Across all queues sizes 16 cores with SMS achieves a higher hit rate than the baseline single core system. This is because SMS is able to reduce the number of conflicts in the request streams of a single core as well as between request streams of different cores. In our 16 core, 32 entry queue configuration, SMS increases the row hit rate to 78% compared to 33% without SMS. However, we still see that there is potential improvement beyond just adding an SMS prefetcher. Our single core, 32 entry queue configuration is able to achieve 86% row hits, indicating some of the locality in the request stream from a single processor is not captured with SMS. Also, our 16 core, 2048 entry queue configuration achieves 90% row hits, indicating that there are still potential row hits not being captured because of the size of the queue.
3.2 Timing Simulation

Next we consider the effects of timing on the available row buffer locality by using our detailed timing simulator described in Section 5.4. This timing simulator models a number of real system timing characteristics that affect the access pattern of memory requests. First, it models the delay between consecutive prefetches caused by delays in the L2 and L3 caches. Second, it models the highly variable network delays resulting from different requests going to different slices of the distributed L3 cache. Third, it models the rate at which requests are issued to DRAM from the command queue. These three factors cause the chance of local requests residing in the queue at the same time to be lower than in our previous analysis. With fewer local requests queued in the memory controller, the scheduler has less opportunity to schedule row hits.

Figure 7. Timing results comparing the actual and projected row hit rate for a 16 core system with a 32 entry queue
Figure 7 shows the **actual** row hit rates from our timing simulations for each workload on a 16 core system with 32 entry command queues. These simulations were run with the baseline SMS configuration described in Section 5.1. As discussed above, accurately modelling the timing of requests reveals that the memory scheduler has less opportunity to schedule row hits and results in a lower hit rate. The average hit rate for our 16 core, 32 entry queue system is 43%, compared to 78% in our initial analysis. The difference in these two values highlights the importance of accurate timing simulation to evaluate this work.

We also project how many row hits we can expect from a scheme which forces prefetch bundles to experience the maximum number of row hits. To track prefetch batches we tag each prefetch with a unique batch id when it is issued by the prefetcher. At the memory controller we record the prefetches that arrive for each batch, and whether they result in a row hit. Using a scheduling policy that uses prefetch information to maximize row hits, we should be able to guarantee that every batch of \(N\) prefetch results in \(N-1\) row hits. Note that we cannot guarantee that the first prefetch we issue from the batch will hit in an open row, but once the first prefetch opens the row we can force the rest of the prefetches in that batch to be issued before the row is closed. We do this analysis for our typical 16core-32queue configuration and include this result in Figure 7 as the **projected** hit rate. We expect, on average, that this informed prefetch scheduling scheme will increase the row hit rate by 16% over the basic open page policy.
4 Design

We saw in the previous chapter that some of the row locality presented by SMS is not captured by the memory controller. The primary goal of our design is to capture all of the potential row hits present in batches of SMS prefetches. In order to do this we need to prevent SMS batches from getting interleaved with the request streams of other cores. We do this by bundling the prefetches from a batch together so that the memory controller can issue them together and prevent other requests from interleaving. This guarantees that all the potential row hits from a prefetch batch actually occur. Additionally, by reducing the interleaving of prefetch requests, we indirectly increase the row locality of the demand requests that would otherwise be interleaved with the prefetches, thus reducing the latency of the more critical demand requests.

We also want to be able to filter our prefetch bundle for cache hits. If a prefetch bundle finds a requested block of data in the cache then it no longer needs to access that data in memory. As such we want to remove that prefetch from the bundle. Filtering the bundle incurs some overhead that we discuss in Section 4.1. The alternative is to not update the bundle, speculatively prefetch the entire bundle from memory, and drop the redundant prefetches. However, the overhead of additional off chip and network traffic outweighs the overhead of filtering the prefetches.

In Section 4.1 we describe two methods of bundling prefetches together and justify our choice of the Tagged Bundle method. In Section 4.2 we explain the design of the tagged bundle in more detail. In Section 4.3 we describe the implementation details of our design. In Section 4.4 we summarize the selected parameters for our design. In Section 4.5 we describe a scheduling policy designed to work with bundled prefetches. In Section 4.6 we describe an optimization enabled by out scheduling policy. Finally in Section 4.7 we describe a deadlock scenario inherent in our design and how we avoid it.
4.1 Single Packet Bundling vs. Tagged Bundling

There are two fundamentally different ways to bundle prefetches that were considered for this work. The intuitive approach is to compress all of the prefetch requests into a single packet and transmit that packet through the memory subsystem. However, this approach has some significant drawbacks which we describe in Section 4.1.1. Instead we propose a Tagged Bundle approach where each prefetch in the bundle is tagged such that the memory controller can reassemble the prefetch bundle. We describe this approach, its advantages and disadvantages in Section 4.1.2.

4.1.1 Single Packet Bundle

A prefetch bundle is identified by two pieces of information, the spatial region address and the spatial pattern. As described in Section 2.1.1, the spatial region address is the higher order bits that are common to each prefetch address in a bundle. The spatial pattern is the bit vector that indicates which of the blocks in the spatial region are being prefetched. We can create a single packet that contains the spatial region address and the spatial pattern which defines all of the prefetches in the bundle. By compressing the individual prefetch requests into a single packet we can reduce the request traffic in our system.

Filtering cache hits from the prefetch bundle incurs additional delay. Before the packet can be issued from the cache the spatial pattern must be updated for every cache hit. This means that the bundle must wait at each level of cache for each prefetch to perform a tag look-up. In the worst case this requires 31 tag look-ups, and as we show in Section 5.1, 31 prefetch bundles make up 23% of all bundles. For single bank caches this will require 31 serial tag lookups, while multi-banked caches will be able to partially parallelize these lookups [30].

A common configuration for the last level cache (LLC) is a distributed, statically mapped shared cache [10]. In this configuration the cache is divided into banks and the banks are distributed across all the nodes in the interconnect network. Cache blocks are mapped to banks by their lower order bits, so that consecutive cache blocks map to different banks. This allows...
sequential requests to be serviced in parallel by the LLC and increases request bandwidth. In order to support Single Packet Bundles for such an LLC we would either need to use a coarser grained mapping where each spatial regions map to the same bank, which would reduce cache bandwidth, or potentially visit multiple cache banks per bundle, which would increase prefetch latency.

Single Packet Bundles offer lower request traffic but will add serialization delay at each level of cache and negatively impact either latency or bandwidth of the LLC.

4.1.2 Tagged Bundle

To avoid the complexities introduced by Single Packet Bundling we also consider a Tagged Bundle approach. Tagged Bundles transmit prefetches as individual requests, but include information so the memory controller can reassemble the bundle. The memory controller buffers the prefetches from a bundle until all expected prefetches arrive.

On a cache hit we again want to remove that prefetch from the bundle. However, the bundle information is now stored at the memory controller. Therefore we must send a separate prefetch hit notification message must be sent to the memory controller to update it. This prevents the memory controller from waiting for a prefetch that will never arrive and unnecessarily delaying the rest of the bundle.

Instead of waiting for all the prefetches to arrive, the memory controller has the option to issue the buffered prefetches to trade off some of the row locality for lower prefetch latency. With Single Packets Bundles, all the prefetches experience the same delayed arrival. Therefore Tagged Bundles offer more flexibility to the memory scheduler.

Tagged Bundles increase network traffic by increasing the size of prefetch requests and generating hit notifications, while Single Packet Bundles reduce network traffic by compressing prefetch bundles into a single packet. However, Tagged Bundles do not directly increase the latency of prefetches in the cache hierarchy and are not confined to sub-optimal configurations of the LLC. Tagged Bundles also allow for more sophisticated memory scheduling policies. As such, we chose to use Tagged Bundles over Single Packet Bundles.
Figure 8 illustrates the timing of Single Packet Bundles and Tagged Bundles in the cache hierarchy. In Figure 8a), all the prefetches arrive together at $t_{0.3}$. In Figure 8 b) the first prefetch arrives at $t_0$, well ahead of $t_{0.3}$, at which point the memory controller could choose to issue it. The last prefetch arrives at $t_2$ and the last notification arrives at $t_3$. At $t_3$ the memory controller has full information about the bundle, earlier than with Single Packet Bundling.
Figure 8. Timing diagram for a) Single Packet Bundles and b) Tagged Bundles
4.2 Prefetch Bundling Design

We propose a system that bundles prefetches by tagging each prefetch with information about the other prefetches in the bundle. Bundled prefetches are filtered through the caches, where each cache hit sends a notification to the memory controller. The memory controller collects each bundle and issues the prefetches to memory when the entire bundle has arrived.

We add a new message type to the system, called a Bundled Prefetch, which acts like a normal prefetch except it also contains the additional bundling information. We distinguish this from a normal prefetch because we want to be able to use normal prefetches for batches that contain a single prefetch. A one prefetch bundle would offer the memory controller no spatial locality to exploit, but would still incur the overheads of a Tagged Bundle. As such, we reserve normal prefetch requests for these one-prefetch batches.

Each bundled prefetch is tagged with two pieces of information, a bundle ID and a spatial pattern. Figure 8a shows an example of four bundled prefetch requests with these additional fields.

The bundle ID is a unique number that identifies the bundle within the system. The bundle ID is made up of two numbers, the core ID, identifying the requesting core, and the bundle index, identifying the bundle among in-flight bundles from the same core. We impose a limit on the number of bundles that a core can have in flight at the same time. As such we can easily reuse the bundle index of completed bundles. For example, in a system with a limit of 16 in flight bundles, we can always select a bundle index from 0 to 15, making the bundle index 4 bits. So for a 16 core system with a 4 bit core ID, the bundle ID would require only 8 bits.

The spatial pattern tells the memory controller which other prefetches were issued as part of the prefetch bundle. This is essentially the spatial pattern that is stored in the prefetcher's PHT that was used to generate the prefetch batch. The spatial pattern tells the memory controller which prefetches are in the bundle so the memory controller knows whether or not it should wait for more prefetches to arrive before issuing the bundle to memory.

To keep track of the prefetch bundles at the memory controller, we add a new structure called the prefetch bundle queue. Entries in the queue are identified by their bundle ID and
record all the prefetches that have arrived at memory and all the prefetches that are expected to arrive in the future. In order to avoid deadlock, we use a prefetch bundle queue that is large enough to hold all the in-flight prefetch bundles in the system. Since we limit the number of in-flight prefetch bundles per core we can meet this constraint with a finite sized queue. We discuss deadlock avoidance in more detail in Section 4.7.

4.3 Prefetch Bundling Implementation

To support Prefetch Bundling we need to modify a number of components in our system. The prefetch controller needs to be modified to tag the bundled prefetches and limit the in-flight bundles. The caches and interconnect must be modified to support prefetch hit notifications. Finally at the memory controller we need to add a prefetch bundle queue to reassembly the bundles and buffer them for the scheduler.

4.3.1 Prefetch Controller

To support Prefetch Bundling, the SMS prefetch controller needs to determine the bundle ID and spatial pattern for each prefetch request. The spatial pattern comes directly from the SMS pattern history table. However, the prefetches generated by the PHT are filtered to prevent duplicate prefetches, so the spatial pattern must be updated as well. In order to do this we need to delay the filtered prefetches until the entire batch has been filtered and the final spatial pattern is known.

The bundle ID is comprised of the core ID and the bundle index. Determining the core ID is trivial. To assign the bundle index we need to track which bundles are currently in flight and choose a bundle index that is not currently in use. Tracking in flight bundles is done with an array of counters, one for each bundle index and a 16 bit register, where each bit indicates whether the corresponding index is in use. We call this array of counters and bit vector the Bundle Tracker. When a bundle is issued, its counter is set to the size of the bundle and the corresponding bit in the index register is set to 1. Each time we receive a bundled prefetch reply, the counter is decremented. If the counter reaches zero then the corresponding bit in the index register is set to 0. When a new bundle is issued, the index register is scanned and the index of
the first 0 is selected as the bundle index. Figure 9 shows the SMS backend modified to support Prefetch Bundling. It shows the state of the Bundle Tracker that generated the bundle index, before being updated to account for the newly issued bundle.

![Diagram of Modified SMS Backend]

**Figure 9. Modified SMS backend**

### 4.3.2 Caches and Interconnect

To support Prefetch Bundling, the cache requires minor modification to generate prefetch hit notifications. If the request is a bundle prefetch and the look up resulted in a hit, then the cache should generate a prefetch hit notification. A prefetch hit notification message is identical to the bundled prefetch request that would be generated on a miss, except it has a different message type.

The routers in the interconnect network need to be modified to route prefetch hit notifications from the L2 directly to the appropriate memory controller node, instead of sending them to the L3 like a prefetch request.
4.3.3 Bundle Queue

As mentioned in Section 4.2, we size the bundle queue to hold all the in-flight bundles in the system. Since we have a maximum number of in-flight prefetch bundles per core, we can calculate the bundle queue size as shown in Equation 1.

**Equation 1**  \( \text{Bundle Queue Size} = \text{Max prefetches per core} \times \text{Number of cores} \)

Since each prefetch bundle gets a globally unique bundle ID, we can make the prefetch bundle queue a direct mapped structure which is indexed by the bundle ID, as shown in Figure 10. Each entry in the prefetch bundle queue contains these three fields:

**Spatial region address.** The higher order bits off the address which are common to all blocks in the spatial region.

**Present vector.** Like the spatial pattern except it only has bits set for prefetches that have reached memory and are currently being queued by the prefetch bundle queue.

**Expected vector.** The expected vector indicates all the prefetches from that bundle that are expected to reach memory.

![Figure 10](image.png)

**Figure 10.** Modified memory controller with Prefetch Bundle Queue
When the first prefetch from a bundle arrives at memory it selects the entry corresponding to its bundle ID and sets the fields accordingly:

- The spatial region address is extracted from the higher order bits of the prefetches address
- The present vector is set to 1 for the bit corresponding to the spatial region offset, 0 everywhere else
- The expected vector is copied from the spatial pattern field of the prefetch.

If a prefetch hit notification is the first message to arrive for a bundle then it will also allocate in the bundle queue. It initializes all the fields as described above except in both the present vector and expected vector it sets the corresponding bit to 0.

When a subsequent prefetch arrives at memory it updates the present vector by setting the corresponding bit. When a prefetch hit notification arrives, the corresponding bit in the expected vector is set to 0. When a prefetch is issued from the prefetch bundle queue then the corresponding bit in both the present vector and expected vector is set to 0.
4.4 Bundling Parameters

We calculate the bundle queue size using the relevant system parameters used in our evaluation in Chapter 5. We summarize the system parameters in Table 1 and the derived bundling parameters in Table 2. We assume a 40-bit address space so a 2 kB spatial region has a 29 bit spatial region address.

<table>
<thead>
<tr>
<th>Table 1. SMS and system parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial region size</td>
</tr>
<tr>
<td>Cache block size</td>
</tr>
<tr>
<td>Max. in flight bundles per core</td>
</tr>
<tr>
<td>Number of cores</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2. Prefetch Bundling parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bundle queue size</td>
</tr>
<tr>
<td>Entry size</td>
</tr>
<tr>
<td>Bundle Queue storage requirements</td>
</tr>
<tr>
<td>Bundle tag size</td>
</tr>
</tbody>
</table>

Using CACTI as described in Section 5.5 we estimate an access time of 0.254 ns. This is well within the memory controllers cycle time of 0.313 ns so the bundle queue can be accessed in a single cycle.

4.5 Scheduling Policy

The scheduling policy can be divided into two components, transaction scheduling and command scheduling. Transaction scheduling decides which transaction in the transaction queue gets converted into commands and sent to the command queue. Command scheduling decides
which commands in the command queue get issued to memory. In this work we use the simple First Read First Come First Served (FR-FCFS) policy for command scheduling and design a transaction scheduling policy to leverage the prefetch bundle queue.

We employ a simple transaction scheduling policy aimed at maximizing row hits. Prefetches are issued from the prefetch bundle queue on one of two conditions, all expected prefetches in a bundle are present or the corresponding row is already open. Each time a bundled prefetch is received the prefetch bundle queue entry is updated and if the present vector is equal to the expected vector then all of the prefetches in the bundle are issued to the command queue. In doing so, we guarantee that for an N-prefetch bundle we will get at least N-1 row hits.

Each time a transaction is issued to the command queue, either from the transaction queue or the bundle queue, we check the bundle queue for a bundle going to the same row. If we find a bundle going to the same row then we issue all of its present prefetches.

4.6 Tagging Optimization

For simpler scheduling policies like the one we present in Section 1.5 we optimize the way we record bundle information. The key observation is that for our scheduling policy we do not need to know which blocks we are expecting prefetches for, rather we just need to know how many blocks. As such, we can replace the spatial pattern in the bundled prefetch and the expected vector with bundle sizes instead.

To support this we make the following changes to our system. The prefetcher has to count the 1s in the filtered spatial pattern and store this count, the bundle size, using 5 bits in the prefetch tag. In the bundle queue we replace the expected vector with a 5 bit expected bundle size field. When a bundle allocates a new entry in the bundle queue, the bundle size is copies to the expected bundle size field. If the first message is a prefetch hit notification then the bundle size is decremented before being stored. On subsequent prefetch hit notifications the expected bundle size is decremented. Each time an entry is updated, the number of 1s in the present vector and the expected bundle size are compared. When they are equal the bundle is issued to the command queue.
The optimized bundling parameters are summarized in Table 3. Compared to the original bundling parameters in Table 2 we are able to reduce the tag size by 51% and the bundle queue storage requirements by 32%.

**Table 3. Optimized Prefetch Bundling parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bundle queue size</td>
<td>256 entries</td>
</tr>
<tr>
<td>Entry size</td>
<td>32+5+29=66 bits</td>
</tr>
<tr>
<td>Bundle Queue storage requirements</td>
<td>2112 Bytes</td>
</tr>
<tr>
<td>Bundle tag size</td>
<td>5+8=13 bits</td>
</tr>
</tbody>
</table>

### 4.7 Deadlock Avoidance

In this section we describe the motivation for making the prefetch bundle queue large enough to hold all prefetches. In Section 1.4 we describe a simple scheduling policy that waits for all prefetches to arrive before issuing the bundle to memory. With a smaller prefetch bundle queue this simple scheduling policy can lead to deadlock when a bundled prefetch tries to allocate in when the queue is full. This is different from a normal cache or queue conflict scenario because the bundle queue does not just hold the pending bundled prefetches, but also a record of the global state of each bundle. If a bundle either bypasses the bundle queue or is evicted from the bundle queue then this global state record is lost and subsequent prefetches from the bundle can allocate in the queue with incomplete information about the state of the bundle.

Consider a new bundle arriving at the memory controller with a full prefetch bundle queue. There are three ways to deal with this, each of which leads to a deadlock.

1. Stall the new request and wait for a slot in the bundle queue to become available. However, all the slots are waiting for bundled prefetches that will end up queued behind the new request so no slots will open up.
2. Bypass the bundle queue and issue the new request to the transaction queue. Subsequent prefetches from the same bundle that get allocated in the prefetch bundle queue will wait for the first prefetch, since there is no record that it was already processed.

3. Evict a bundle from the queue by issuing all the present prefetches in that bundle and clearing the entry. Again, there will be no record that these prefetches were processed and subsequent prefetches from the same bundle will allocate in the prefetch bundle queue and wait for the already processed prefetches.

We consider two ways to deal with these deadlock scenarios.

1. Add timestamps to each entry and impose a time limit for bundles to exist in the queue.

2. Avoid conflicts by making the queue large enough to hold all prefetches in the system.

Option 1 allows us to have smaller queues and save on area costs. However, in the event of a conflict the system has to wait for a timeout to resolve the deadlock. Also consider that since an incomplete bundle is being evicted on a timeout, the rest of that bundle will subsequently allocate in the queue and wait there until a timeout since the original bundle information was lost. So each conflict will cause two timeouts.

Option 2 will have higher area cost but will not have a direct impact on performance. As mentioned in Section 4.4, a maximum sized queue can still be accessed in a single cycle so the increased queue size will not incur extra latency.

We chose the higher performance option of the maximum sized queue. However, the relative area to performance trade-off is not clear in our high level analysis. We leave the evaluation of a smaller queue with timeouts as future work.
5 Methodology

In order to evaluate our Prefetch Bundling scheme we use Flexus[4] in combination with WindRiver Simics[13] to accurately model a 16-core processor. The system parameters are described in Section 5.1. We use a mix of parallel and commercial server workloads described in Section 5.2. To motivate this work we used a trace based simulation methodology to analyze memory traces, described in Section 5.3. In Section 5.4 we describe how Flexus, Simics and DRAMsim were used to accurately evaluate our system. In Section 5.5 we describe the tools used to calculate the area and power costs of our design.

5.1 System Overview

Table 4 summarizes the parameters of our simulated chip multiprocessor. We model a 4 GHz out of order processor designed for the UltraSPARC III ISA. It uses a split L1 instruction and data cache and an inclusive, private L2 at each core. It also has a shared L3 cache which is divided into 16 banks and distributed across the chip with one bank per core. It maintains cache coherence with a directory and a MESI coherence protocol. The cores are connected with a mesh interconnect network which uses X/Y routing.
Table 4. CMP parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Core</td>
<td>16 core, 4 Ghz UltraSPARC III ISA, 8-stage out-of-order pipeline, 128-entry ROB, 64-entry LSQ, decode/issue/commit any 4 instrs/cycle</td>
</tr>
<tr>
<td>Branch Predictor</td>
<td>8K Gshare, 16K bi-modal, and 16K selector 2K entry, 16-way BTB, 2 branches/cycle</td>
</tr>
<tr>
<td>Fetch Unit</td>
<td>Up to 8 instrs/cycle, 32-entry fetch buffer</td>
</tr>
<tr>
<td>L1 I+D</td>
<td>64 kB, 64 B blocks, 2 way, 2 cycle</td>
</tr>
<tr>
<td>Private L2</td>
<td>256 kB, 8-way, inclusive of L1, 2/5-cycle tag/data</td>
</tr>
<tr>
<td>Shared L3</td>
<td>16 MB, 16 banks, 16 way, non-inclusive, 3/9-cycle tag/data</td>
</tr>
<tr>
<td>Cohererce</td>
<td>MESI protocol, directory based, 4 cycle lookup</td>
</tr>
<tr>
<td>Interconnect</td>
<td>4 x 4 Mesh, X/Y Routing, 128 bit links, 3 virtual channels, 3 cycles per hop</td>
</tr>
</tbody>
</table>

Table 5 summarizes the parameters used for our SMS prefetcher, both in the baseline SMS system and our proposed Prefetch Bundling system. These parameters were chosen based on the evaluation done in the original SMS work [19]. Since we are using many of the same workloads we expect the conclusions of the SMS paper to hold.

Table 5. SMS parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial Region Size</td>
<td>2 kB (32 cache blocks)</td>
</tr>
<tr>
<td>Filter Table Size</td>
<td>32 entries</td>
</tr>
<tr>
<td>Accumulation Table Size</td>
<td>64 entries</td>
</tr>
<tr>
<td>PHT Size</td>
<td>16k entries, 16 way</td>
</tr>
<tr>
<td>PHT indexing</td>
<td>Rotation indexing</td>
</tr>
<tr>
<td>PHT update</td>
<td>2 bit saturating counters</td>
</tr>
</tbody>
</table>
Table 6 summarizes the memory parameters used in our system. Our memory device parameters are taken from the data sheet for the Micron MT41J256M4 1600MHz DDR3 dram chip [14]. We use 16 chips per rank and two ranks per channel. Each memory controller in our system connects to one memory channel. In our evaluation we vary the number of memory controllers between 1 and 4. This gives us a total memory capacity of 4 to 16 GB.

To optimize for increased row hit rate we use the row-rank-bank-channel-column mapping scheme and FR-FCFS scheduling [17]. This mapping scheme prioritized row buffer locality over parallelism by using the least significant bits to select the column and the higher bits to select the rank, bank and channel. FR-FCFS is used in all configurations to prioritize row hits available in the command queue.

<table>
<thead>
<tr>
<th>Table 6. DRAM parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM device parameters</td>
</tr>
<tr>
<td>Micron MT41J256M4 DDR3-1600 11-11-11 Timing parameters [14]</td>
</tr>
<tr>
<td>Device size</td>
</tr>
<tr>
<td>8 banks *16k columns * 2k columns * 4 bits/column = 1 Gb</td>
</tr>
<tr>
<td>Rank size</td>
</tr>
<tr>
<td>1Gb devices * 16 devices/rank = 2 GB</td>
</tr>
<tr>
<td>Channel size</td>
</tr>
<tr>
<td>2 GB rank * 2 ranks/channel = 4GB</td>
</tr>
<tr>
<td>Channels</td>
</tr>
<tr>
<td>1 channel/memory controller, 1/2/4 memory controllers</td>
</tr>
<tr>
<td>Address mapping</td>
</tr>
<tr>
<td>Row-rank-bank-channel-column</td>
</tr>
<tr>
<td>Memory controller clock</td>
</tr>
<tr>
<td>3.2 GHz</td>
</tr>
<tr>
<td>Transaction queue</td>
</tr>
<tr>
<td>32 entries</td>
</tr>
<tr>
<td>Command queue</td>
</tr>
<tr>
<td>32 entries, one queue per rank</td>
</tr>
<tr>
<td>Command queue scheduling</td>
</tr>
<tr>
<td>FR-FCFS</td>
</tr>
</tbody>
</table>
5.2 Workloads

We select workloads from a variety of benchmark suites. We organize them in to three categories which we refer to as commercial[1], PARSEC[2] and scientific[26]. Table 7 summarizes the workloads and their configurations. The commercial and scientific workloads were chosen as a selection of the workloads used in the original SMS paper to allow us to compare our results with the original paper. The two PARSEC workloads where chosen for having dense prefetch batches.

Table 7. Workloads

<table>
<thead>
<tr>
<th>Suite</th>
<th>Description</th>
<th>Name</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Commercial</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPECweb99</td>
<td>Web Server</td>
<td>apache</td>
<td>16K connections, FastCGI, worker threading</td>
</tr>
<tr>
<td>TPC-C</td>
<td>Online Transaction Processing</td>
<td>tpcc</td>
<td>DB2, 16 clients, 100 warehouses (10 GB)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oracle</td>
<td>64 clients, 100 warehouses (10 GB), 450MB buffer pool</td>
</tr>
<tr>
<td>TPC-H</td>
<td>Decision Support</td>
<td>tpch</td>
<td>DB2, commercial database system, 450 MB buffer pool</td>
</tr>
<tr>
<td><strong>PARSEC</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARSEC 2.1</td>
<td>Financial calculation</td>
<td>blackscholes</td>
<td>native input</td>
</tr>
<tr>
<td></td>
<td>Similarity search</td>
<td>ferret</td>
<td>native input</td>
</tr>
<tr>
<td><strong>Scientific</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPLASH-2</td>
<td>Electromagnetic force simulation</td>
<td>em3d</td>
<td>600k nodes, degree 2, span 5, 15% remote</td>
</tr>
<tr>
<td></td>
<td>Current simulation</td>
<td>ocean</td>
<td>1026 × 1026 grid, 9600 seconds</td>
</tr>
<tr>
<td></td>
<td>Cholesky factorization</td>
<td>sparse</td>
<td>4096 × 4096 matrix</td>
</tr>
</tbody>
</table>
5.3 Trace Based Simulation

We use a simple, two phased approach to quickly collect measurements to motivate our work. First we run a functional Flexus simulation to generate memory traces. Then we feed these memory traces into a custom memory simulator to analyze the memory access pattern. We track row accesses with this simulator to determine the row hit rate of each memory trace.

Functional simulation is done over 2 billion cycles to generate a memory request trace for each workload. Simics performs functional simulations of the application and operating system. Flexus takes the instruction stream from simics and executes it on a simple functional model of our 16 core architecture. In functional simulation loads and stores propagate through the cache to memory and return a reply in a single cycle. In functional simulation we do not model the interconnect network and the private L2 caches are connected to memory via a single shared L3 cache. In this way we are only able to generate traces that capture the approximate order of requests as they arrive at memory but not the timing of their arrival.

Traces are then fed into a simple custom functional memory simulator. This simulator maps each address to the corresponding DRAM channel, rank and bank according to our memory mapping scheme and uses open page row management. For each bank within each rank and channel we track which row is currently open. Traces are fed into a queue of variable size. Since there is no timing information in the traces, we maintain a full queue at all times. In this way the queue behaves less like a real memory queue, whose occupancy would be determined by the rate at which requests arrive and are serviced, and more like a window of future requests. With this single queue we implement FR-FCFS to select requests to issue to memory. We count the number of row misses and row hits to determine the row hit rate.

We measured row hits with three queue sizes. We use a queue size of 1 to measure the inherent row locality of the memory request trace without any reordering of requests. A queue size of 32 was used to approximate the amount of reordering you would see in a real system with a reasonably sized 32 entry transaction queue. A queue size of 2048 was used to put a reasonable upper bound on the potential row hit rate we could achieve.
5.4 Timing Simulation

The majority of our evaluation is done using detailed timing simulation with Flexus and Simics. We replace the simple functional models of our core and caches with detailed timing models that accurately capture the delays incurred by memory requests in the cache hierarchy and network.

To model memory we use DRAMsim\cite{24}, a detailed memory simulator that is configured with the standard memory timing parameters to model the precise timing of DRAM operation.

We use the SMARTS sampling methodology\cite{28} to reduce the runtime of our timing simulations. SMARTS is a framework which applies statistical sampling theory to determine an optimal set of sample periods from a program's execution to approximate the full execution. We use functional simulation to warmup the caches, directory, branch predictor and prefetcher and save checkpoints of these states at regular intervals. The warm up time and intervals are determined by SMARTS for each workload. Timing simulations are initialized with the state from each checkpoint and run in parallel for 150 000 cycles per checkpoint. For each checkpoint we use the first 100 000 cycles to warmup the cache, network and memory queues with in flight requests and then collect measurements over the last 50 000 cycles. We measure performance with the User Instructions per Cycle (UIPC) of each workload. User IPC is calculated as the number of committed user level instructions over the number of cycles.

5.5 Area and Power Models

DRAMsim also models the power for the DRAM modules. It uses current and voltage parameters given in the DRAM data sheet. It reports the Average Power broken down into Background, Activate/Prefetch, Burst and Refresh power.

We used CACTI\cite{21} to model the additional memory structures used by our design. CACTI is an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. We use the access time reported by CACTI inform the choice of our memory structure sizes. Structures were modelled for a 32 nm technology node. For all structures, we use low standby power transistors for the data array and high performance transistors in the peripheral circuitry.
6 Evaluation

In this chapter we present the evaluation of our proposed Prefetch Bundling scheme. In Section 6.1 we present a characterization of our workloads focusing on their prefetch and memory access characteristics.

6.1 Workload Characterization

In this section we present a characterization of our workloads. The observations made here will be used in our analysis of individual workloads in Section 6.2.

6.1.1 Bundle Size Distribution

Figure 11 shows the bundle size distribution of all prefetch bundles issued for each workload. The leftmost bar shows the percentage of 2-prefetch bundles and the rightmost bar shows the percentage of 31-prefetch bundles. Note that the largest bundle size is 31 prefetches because SMS tracks access patterns in regions of 32 blocks and one of those accesses will be trigger access, which should not be prefetched. Larger prefetch bundles have a bigger impact on the performance of our scheme than smaller bundles. Considering the two extremes, a 31-prefetch bundle guarantees 30 row hits, a row hit rate of 97%, while a 2-prefetch bundle only guarantees one row hit, a row hit rate of 50%.
Figure 11. Prefetch bundle size distribution. The leftmost bar is 2-prefetch bundles, the rightmost bar is 31-prefetch bundles.

The scientific workloads exhibit the highest average bundle density. Em3d exhibits many short strides which span 2-3 cache blocks on average, resulting in a mode bundle size of 16. Sparse and Ocean exhibit primarily scan access patterns so 60-70% of bundles contain 31 prefetches. PARSEC workloads also have a large percentage of scan access patterns. In Ferret the 31-prefetch bundles make up 17% of all bundles and in Blackscholes they make up 47%.

The commercial workloads have an exponential distribution of bundle density all with approximately 35% of bundles containing 2 prefetches. The Database workloads TPC-C, TPC-H, and Oracle exhibit sparse access patterns resulting from queries that only access a few columns.
in tables with very large rows. Apache handles small amounts of data per request so most access patterns only span a few consecutive cache blocks.

From the distribution of bundle sizes we expect the scientific and PARSEC workloads to see the most improvement since we are able to guarantee a higher percentage of row hits for dense bundles.

In this work we are only interested in prefetches that go off chip. However, in our scheme, cache hits require notification messages to be sent to memory. So there is a network traffic overhead associated with all on chip fills. Also, each prefetch bundle must wait for all prefetches or notifications to reach memory before the prefetches are issued.

6.1.2 Prefetch Fill Levels

Figure 12. Prefetch Fill Level Distribution
This can lead to a scenario where all the memory bound prefetches have arrived at the memory controller but our scheme will wait for notifications for the rest of the bundle, delaying the prefetches unnecessarily. More cache hits mean there is a higher chance of this scenario happening and prefetches incurring unnecessary delay.

Figure 12 shows the distribution of fill levels for all prefetch requests across our workloads. The fill level is the level in the memory hierarchy which contains the requested data. We categorize fill levels as L2, L3, peerL1, peerL2, and off-chip. An L2 fill means the data was found in the core’s local L2 cache. An L3 fill means the data was found in the processor’s shared L3 cache. PeerL1 and PeerL2 fills mean the data was found in a private cache of another core. An off-chip fill means the data was not found in any cache and must be retrieved from memory.

The scientific and PARSEC workloads see on average 90% of their prefetches go to memory. Conversely, the commercial workloads see only an average of 40% of prefetches reaching memory. We expect the added overheads of hit notifications from on chip fills to contribute to lower performance for the commercial workloads when compared to the scientific and PARSEC workloads.
6.1.3 Memory request breakdown

![Bar chart showing the number of prefetches as a percentage of total memory requests for various workloads.](image)

**Figure 13. Prefetches as a percentage of total memory requests**

Our goal is to improve the overall row hit rate of memory. However, our scheme only leverages the row locality in prefetch requests. Therefore we expect the impact of our scheme to be correlated with the ratio of prefetches to demand requests serviced by memory. Figure 13 shows the number of prefetches that reach memory as a percentage of the total number of memory transactions. The scientific applications and ferret all generate prefetches that make up over 85% of all transactions. Blackscholes is in the middle with 64% of transactions, while the commercial workloads generate prefetches that make up less than 50% of all memory transactions. Again we expect higher performance improvements from the scientific and PARSEC workloads than from the commercial workloads.
6.1.4 Memory Traffic

![Bar chart showing memory transactions per KI]

Figure 14. Memory Transactions per kilo-instruction

The contribution of our scheme to overall performance depends on the relative amount of memory traffic generated by each workload. A workload that generates more memory traffic will likely spend more time stalled waiting for data from memory and thus be more sensitive to improvements in the memory system.

Figure 14 shows the amount of memory traffic generated for each workload. This gives us an indication of how each workloads performance will correlate with the systems memory performance. On average our workloads generate 29 memory transactions for every thousand instructions executed. Em3d generates significantly more traffic than the other workloads with 147 transactions per thousand instructions. The PARSEC workloads on the other hand are noticeably more compute bound with less than 4 transactions per thousand instructions. Again we expect the scientific workloads to see the most improvement from our scheme. Although the PARSEC workloads have favorable prefetch characteristics, the relatively low memory traffic indicates that improved memory efficiency will have little effect on performance.
6.2 Prefetch Bundling

In this section we evaluate the performance of our Prefetch Bundling scheme. We use two baselines configurations for our comparison denoted as baseline and baseline SMS. The baseline uses the CMP parameters in Table 4 and the DRAM parameters in Table 6 and has prefetching disabled. The baseline SMS configuration adds SMS prefetching with the parameters listed in Table 5. The configuration using our Prefetch Bundling scheme is denoted as bundling and uses the optimized configuration given in Table 1 and Table 3.

6.2.1 Row Hit Rate

To evaluate the performance of our bundling scheme we first look at our target metric of row hit rate. Figure 15 shows the row hit rate for the 2 memory controller configuration for each workload. On average our bundling scheme achieves a 59% row hit rate compared to 54% with a baseline SMS system, an improvement of 11%. However most of that improvement comes from the Sparse workload, in which bundling improves the hit rate by 70%. Without Sparse the average improvement over baseline SMS is only 3%. Sparse sees such a large improvement because prefetches make up a large percent of its memory traffic, it issues mostly dense prefetches and the majority of those prefetches reach memory. The same is true of Ocean but ocean has a higher row hit rate with baseline prefetching, 84% vs 54%, so there is less room for improvement. We expect to see the biggest performance improvement with sparse and the smallest performance improvement with Apache and TPC-H.
Figure 15. Row hit rate for a 2 memory controller configuration

Figure 16 shows how the row hit rate scales with the number of memory controllers. For both configurations the row hit rate increases by 3% when the number of memory controllers are doubled. Adding more memory controllers to the system increases memory level parallelism. So two requests that would have conflicted with each other in a single memory controller system will map to two different channels in a two memory controller system. The improvement gained from bundling also decreases with more memory controllers, from a 12% improvement with 1 memory controller to 8% with 4 memory controllers. The same reason applies here, if there are fewer row conflicts then there are fewer opportunities for bundling to avoid them.
The single memory controller configuration shows the best improvement in row hit rate. However, a single memory controller for a 16 core system is uncommon in practice. 16 core systems will typically have 2-4 memory controllers [18][22][23] to distribute memory traffic on the interconnect network. Going forward we will focus on the 2 memory controller system, but will revisit the scaling from one to four memory controllers in Section 6.2.5.
6.2.2 Memory Latency

Figure 17 shows the average latency of memory requests, the time between a request arriving at the memory controller and the corresponding response leaving the memory controller. Without prefetching requests take an average of 80 ns to be processed by the memory controller. With prefetching, the number of memory requests increases by 33%, as shown in Figure 18. This added memory traffic results in a 78% increase in the average request latency.

![Figure 17. Average request latency. 2 memory controllers.](image)
Figure 18. Memory requests per 1000 cycles. 2 memory controllers.

With Prefetch Bundling we see a 55% latency increase over baseline prefetching. This latency is largely from prefetch requests that are delayed in the bundle queue. Workloads like Blackscholes, ocean and sparse experience the largest increase in latency because they issue mostly dense prefetch bundles. Prefetches from the dense bundles spend more time in the bundle queue waiting for all the prefetches to arrive before being issued. Memory traffic only increases by 2% over baseline prefetching, so its impact on the increased latency is negligible.
Figure 19 breaks down the latency for baseline SMS and bundling into the average latency of demand and prefetch requests. As we expect, the average latency of prefetches increases with bundling because they are delayed in the bundle queue. However, the average latency of demand requests also increases. Workloads with high prefetch bundle density, namely Blackscholes, Ocean and Sparse, exhibit high demand latency increases. Since we prioritize prefetches from completed bundle queue entries, demand requests experience long delays waiting for all of the prefetches from dense bundles to be issued. This motivates the need for a scheduling that enforces fairness to prevent demand requests from experiencing excessive delays. We leave the design of this scheduling policy to future work.
6.2.3 Memory Bandwidth

By delaying prefetch requests for improved row hit rate we are trading off higher request latencies for higher memory bandwidth. Figure 20 shows the bandwidth per channel across all workloads. On average bundling is able to achieve only a 2% performance improvement over the baseline SMS system. Virtually all of that improvement comes from Sparse, which has a 20% increase in bandwidth. The rest of the workloads see no significant increase in bandwidth. This is similar the row hit rates shown in Figure 15 where Sparse has a 70% improvement over the baseline prefetching, while the other workloads have less than 6% improvement. Since no other workloads show significant improvement in either of these metrics we cannot make any strong conclusions about the correlation between row hit rate and bandwidth with our bundling scheme.
6.2.4 Memory Power Consumption

Another area where we benefit from increased row hits is in DRAM power consumption. A significant percentage of DRAM power is spent opening and closing rows so increasing the row hit rate can offer significant power savings.

Figure 21 shows the total power consumption of the DRAM in our system. SMS results in a 3% power increase over the baseline system with no prefetching. This increase is due to the 33% increase in memory traffic from prefetching shown in Figure 18. Normalizing power over the number of transactions serviced SMS provides a DRAM power savings of 43%. Comparing the baseline SMS and our bundling scheme with the same normalization we see that bundling only provides an additional 2% power savings.

![Figure 21. Memory power consumption. 2 memory controllers.](image)
Figure 22. Memory power breakdown. 2 memory controllers.

Figure 22 shows the power consumption breakdown averaged across our workloads. Power consumption is divided into four categories. Refresh is the power consumed by the periodic refresh of the dram array. Burst is the power consumed by transmitting data over the memory bus. Array is the power consumed by array accesses, including both precharge and activate commands. Background is the static power consumption of the row buffer when holding an open row. Since we are using an open row policy the row buffer is always holding an open row and the background power is constant across all three configurations. Refreshes occur at fixed intervals so the power consumption is also constant. The array power is directly related to row hit rate, since each row miss results in one precharge and activate operation. The Array power makes up 28% of the total DRAM power in the baseline configuration with no prefetching. SMS prefetching reduces the Array power to 22% and bundling reduces it further to 21% of total DRAM power. However, the increased number of transactions increases the burst power from 27% in the baseline system to 35% with SMS and 36% with bundling. Since SMS already causes a significant decrease in Activate/Precharge power and a significant increase in burst power, the opportunity for bundling to save power is diminished.
6.2.5 System Performance

Figure 23 shows the relative performance for our baseline SMS system and our bundling scheme normalized to a baseline system without prefetching. For completeness we have included results for 1, 2 and 4 memory controller systems.

First we will examine the performance of our bundling scheme over the baseline SMS prefetching. Figure 23 a) shows the performance for a 1 memory controller system. On average we see a 2% speedup across all our workloads. This is mostly due to the 6% speedup from Blackscholes and the 8% speedup from sparse. Again we see that the workloads that issue dense bundles see the most improvement from bundling. The rest of the workloads show no significant speedup. Figure 23 b) shows the performance of a 2 memory controller system. In this configuration we see no significant speedup from bundling over the baseline SMS prefetching. Figure 23 c) shows the speedup for a 4 memory controller system. In this configuration we see a slowdown of 5% on average over the baseline SMS prefetching. In the 4 memory controller system we see fewer additional row hits from bundling but still pay a high request latency penalty from delaying prefetches in the bundle queue. In this case the benefit of row hits is outweighed by the latency penalty and performance suffers.

Next we consider the overall speedup over the baseline system with no prefetching. With one memory controller we see no significant speedup from either the baseline SMS or bundling systems. With 2 and 4 memory controllers we see speedups of 13% and 21%, respectively. The speedups we measure are significantly lower than the speedups reported in the original SMS paper even though we are using a similar configuration and many of the same workloads. We explore this issue further in Section 6.2.6.

This highlights one of the fundamental trade-offs of this system with respect to the number of memory controllers. SMS performs best with a large number of memory controllers to handle the load that the extra prefetch traffic puts on the system. On the other hand, bundling offers the most improvement when there are few memory controllers and row conflicts are more frequent.
Figure 23. Relative UIPC for 1, 2, and 4 memory controller systems
6.2.6 SMS Performance

The other issue we address is the performance of SMS in the 4 memory controller system. We have already discussed why SMS has poor performance with fewer memory controllers. However, SMS was originally evaluated with 4 memory controllers and many of the same workloads and was reported to have much higher speedup. Figure 24 shows the misses of the L1 data caches in our SMS normalized to the misses of the baseline system with no prefetching. On average we see a 59% increase in the number of misses with SMS prefetching. This indicates that SMS is polluting our L1 cache, kicking out more useful data than it is bringing in. This cache pollution could be alleviated by implementing a prefetch buffer. A prefetch buffer is a separate storage structure to hold prefetched data until it is requested by the processor, thus preventing cache pollution. The original SMS paper [19] does not describe whether a prefetch buffer was used but our results suggest that a prefetch buffer is necessary for their reported speedup. Using a prefetch buffer requires non-allocating prefetch requests, which are not currently supported by our simulator. Adding non-allocating prefetches would require significant effort, so we leave this for future work.

**Figure 24.** L1D Miss rate normalized to the baseline. 4 memory controllers.
7 Related Work

Exploiting row buffer locality is a common objective in memory controller design. Micro-pages [20] use memory remapping to deal with the low row buffer locality of interleaved memory requests from multiple cores. Prefetching is also a common way to increase the row buffer locality in memory accesses. ASD [6] uses memory side prefetching to increase useful row accesses. SRP [12] and PADC [11] coordinate core side prefetchers with memory to balance row buffer hits and fairness. Footprint cache [8] uses SMS prefetching to transfer data between main memory and die-stacked DRAM with high row buffer locality. Complexity Effective Memory Access Scheduling [29] arbitrates memory streams in the interconnect network to reduce interleaving of independent thread requests.

7.1 Micro-pages

Sudan et al. [20] examines the degradation of row buffer locality from the increasing core count in modern CMPs. They propose grouping cache blocks into smaller than dram page sized regions called micro-pages and remapping these micro-pages to put frequently accessed pages in the same DRAM row. Remapping can either be done in software or hardware. In software, OS pages are shrunk to the size of micro-pages so that the remapping can be done through the page table. Hardware remapping is done via a 4096 entry fully associative mapping table that requires 11 KB of storage.

Like Prefetch Bundling, micro-pages is able to increase the row hit rate beyond that of FR-FCFS. However, its hardware implementation requires an 11KB fully associative structure compared to our <2KB direct mapped structure, which will increase its energy and area cost.
7.2 Adaptive Stream Detection

Hur et al. [6] propose Adaptive Stream Detection (ASD), a memory side prefetcher to predict and prefetch sequential streams into a local prefetch buffer. Spatial locality is observed at the memory controller by creating a stream length histogram for all read requests. A statistical model is used to determine how many sequential blocks ahead of the current demand request to prefetch. Prefetches are then issued to memory with lower priority than demand requests.

As a memory side prefetcher, ASD is orthogonal to a processor side prefetcher like SMS. However, the two in combination offer little mutual benefit as SMS supports complex access patterns and ASD is limited to sequential streams. ASD could be configured to guarantee prefetch streams experience a maximum row hit rate. However, ASD has lower coverage than SMS, resulting in a more demand requests to memory which dilute the benefit of prefetch row hits.

7.3 Scheduled Region Prefetching

Lin et al. [12] propose Scheduled Region Prefetching (SRP) which co-ordinates prefetching with the memory controller. They issue prefetches when the DRAM channels are idle and prioritize row buffer hits at the memory controller. The proposed system uses scheduled region prefetching, where prefetches are generated in regions of 4KB around a cache miss. The prefetcher will prefetch all blocks in a region that are not already in the cache. Cache pollution from prefetching the entire region is reduced by prefetching into lower priority blocks in the LRU order.

Like our baseline SMS system, prefetches are issued in batches from within spatial regions. Since it prefetches all the blocks from a region it will get a higher average row hit rate and better coverage at the cost of higher cache pollution and higher off-chip bandwidth. The off-chip bandwidth and resulting latency or fairness penalty are mitigated by prefetching when the memory channel is idle. Like our Prefetch Bundling scheme, this system achieves near 100% row hit rate for prefetches, but by prefetching to idle channels and avoiding conflicting streams instead of scheduling around conflicting streams.
7.4 Prefetch-Aware DRAM Controllers

Lee et al. [11] propose Prefetch-Aware DRAM Controllers (PADC) a memory controller design that dynamically adapts its scheduling and buffer management based on the predicted usefulness of prefetch requests. Useful prefetches get the same priority as demand requests and prefetches to open rows will be services before conflicting demand requests, in essence, FR-FCFS. Less useful prefetches can be dropped in periods of high contention. Prefetch accuracy is measured at each core and prefetch usefulness is determined at the memory controller based on the requesting core.

Like Prefetch Bundling, PADC uses additional information from a core side prefetcher to schedule prefetch and demand requests. With this information it will change how aggressively it schedules prefetches to increase row hits. At most, PADC can chose to always use FR-FCFS and match the row hit rate of our baseline SMS system. As we show, our scheduler in combination with Prefetch Bundling provides additional row hits.

7.5 Footprint Cache

Footprint Cache [8] is a cache design for very large, e.g. 512 MB, last level die-stacked DRAM caches. It allocates blocks at the granularity of DRAM pages to keep the tag array small and fast. However it only stores the blocks it needs to reduce the off-chip traffic required to fetch the blocks from main memory. It uses SMS to predict which blocks in each page will be accessed and prefetches them on allocation of the cache page. It also uses an SMS like spatial pattern to track the valid blocks in each cache page.

Like our baseline SMS system, footprint cache achieves high row buffer locality by sending batches of prefetches to the same DRAM row. Since prefetch batches are issued from the LLC instead of the L1, we expect the batches to arrive at the memory controller closer together in time since they are not delayed by intermediate cache lookups or distributed to different banks of the L3 cache. As a result they should experience less interleaving and achieve a higher row hit rate than if they were issued from the L1. However, the footprint cache tag arrays are distributed across the chip so prefetch batches still have to share the network with
other memory request streams and can still experience interleaving. We expect footprint cache to have higher row locality than our baseline SMS system, but lower row locality than our proposed since it does not guarantee row hits.

7.6 Complexity Effective Memory Access Scheduling

Yuan et al. [29] examines request stream interleaving in Graphical Processing Unit (GPU) architectures. This work aims to reduce the area cost of the memory controller using aggressive out-of-order scheduling, FR-FCFS, by handling the de-interleaving in the interconnect network. It identifies the round robin packet arbitration as the source of interleaving in the network. It addresses this by modifying the router arbitration scheme to continuously give an input the highest priority as long as the requests are going to the same row. With this approach and an in-order memory scheduler the system is able to achieve 91% of the performance of FR-FCFS with a less complex logic.

This work targets GPUs with more cores, more memory controllers and different thread behaviour than our CMP system. This work identifies the network arbitration as the primary source of interleaving. However, in our system the addition of a distributed L3 cache also contributes to interleaving. This work also uses FR-FCFS as an upper bound on performance where as we use it as a baseline.
8 Conclusion

In this thesis we show that the increasing number of cores in a CMP generates increasingly interleaved memory access patterns and significantly decreases the row buffer locality. We also show that spatially correlated prefetches are able to reduce the amount of interleaving and improve row buffer locality. We propose a low overhead scheme to present the memory controller with information about row buffer locality beyond that which is present in the queue. We also propose a simple scheduling policy to maximize row hits.

In evaluating our system we demonstrate that our scheme increases the row buffer hit rate by 11% compared to a system with SMS prefetching. Unfortunately, this does not translate to any appreciable performance improvement or power saving. We show that the improvement in memory bandwidth is small compared to the increase in latency of both prefetch and demand requests. Clearly, this simple scheduling policy exacerbates the fairness issues observed in FR-FCFS. However, this present many opportunities for future work as we will discuss in Section 8.1.

We also show that the increase in row hit rate of our scheme over the baseline SMS configuration depends on the system’s memory level parallelism. As we increase the number of channels the relative improvement of our scheme decreases. We suggest revisiting this for future scaling trends in Section 8.1.

We also examine an issue with our baseline SMS configuration. We show that our baseline SMS system is not able to match the expected performance based on the results of the original paper [19]. We further demonstrate that our baseline SMS system is experiencing significant cache pollution which is contributing to the performance loss. We propose to remedy the situation with a separate prefetch buffer. However, adding that to our system would be non-trivial and we leave it for future work.
8.1 Future work

Prefetch Bundling presents many opportunities for future work. We show that our simple scheduling policy pays a significant latency penalty for the increased row buffer locality resulting in no significant performance gain. This motivates the need for a policy that balances the row locality provided by Prefetch Bundling with other metrics like bank level parallelism, fairness and bus utilization.

The high latency of prefetches can be addressed in a number of ways. Timestamps can be used for each bundle in the bundle queue to limit the amount of time they are delayed. Large bundles will generally spend more time in the bundle queue than smaller bundles so bundle size may be a sufficient approximation of the time queued. Another approach that ensures N-1 row hits is to begin issuing bundles before all the pending prefetches have arrived and then lock the bank to prevent conflicting requests from interleaving. This can help reduce the prefetch latency at the cost of demand latency.

Another solution to decrease prefetch latency is to perform memory side prefetching on the entire speculative bundle at the arrival of the first prefetch and cache them at the memory controller. The individual prefetches can be replied to immediately, eliminating the DRAM access latency. This will provide lower prefetch latency and also gives the scheduler a larger window to schedule prefetch bursts around demand requests. However, this cache comes with a significant area cost. Also, the memory side prefetching will generates bandwidth and power overheads for prefetches which receive hit notifications.

In Section 6.2.2 we describe how workloads with larger prefetch bundles significantly delay demand requests. This suggests that the scheduler should take in to account the size of the bundle in determining priority over demand requests. One solution is to limit the size of high priority bursts from the bundle queue, similar to Parallelism Aware Batch Scheduling (PAR-Bs)[15], to allow demand requests to interleave, increasing fairness at the cost of row buffer
locality. If the size of guaranteed row hit bursts is limited then we can begin issuing bundles once the number of present prefetches hits this limit, reducing prefetch latency.

As suggested by Minimalist Open Page [9], after issuing a set of prefetches to the same row the row can be closed, predicting that no further row hits will come before a conflict. Since the overall row hit rate is close to 50%, and prefetch bundles have a hit rate of over 50%, then other requests should have a lower than 50% chance to be a row hit. So closing the page after a prefetch bundle would be the statistically better choice.

FR-FCFS is used as an upper bound on row hit rate for many balanced scheduling policies such as Minimalist Open Page, Parallelism Aware Batch Scheduling and Fair Queuing Memory Scheduling [16]. Prefetch Bundling extends the upper bound of potential row hits from all the requests in the queue to also include pending prefetches. As such, Prefetch Bundling can be combined with these existing scheduling policies to combine their fairness with the extended row buffer locality provided by the prefetch bundle queue.

The severity of stream interleaving and row buffer conflicts is a function of both the number of cores and the degree of available parallelism in channels, ranks and banks. If the number of cores scales faster than the available memory level parallelism, then we expect Prefetch Bundling can offer more performance. Future work could re-examine Prefetch Bundling with higher core count CMPs.

There are many opportunities for further improvement of Prefetch Bundling. Our evaluation directly motivates additional cases for our scheduling policy to consider. Additionally, strategies used by existing scheduling policies can be adapted for use with the prefetch bundle queue. If systems continue to scale such that row buffer conflicts become more frequent then we expect Prefetch Bundling to offer more performance. All this considered, we believe Prefetch Bundling is a promising framework for future systems.
References


