Memory Row Reuse Distance and its Role in Optimizing Application Performance

Mahmut Kandemir
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
kandemir@cse.psu.edu

Hui Zhao
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
hzz105@cse.psu.edu

Xulong Tang
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
xzt102@cse.psu.edu

Mustafa Karakoy
Department of Computer Engineering
TOBB ETU
Ankara, Turkey
m.karakoy@yahoo.co.uk

ABSTRACT

Continuously increasing dataset sizes of large-scale applications overwhelm on-chip cache capacities and make the performance of last-level caches (LLC) increasingly important. That is, in addition to maximizing LLC hit rates, it is becoming equally important to reduce LLC miss latencies. One of the critical factors that influence LLC miss latencies is row-buffer locality (i.e., the fraction of LLC misses that hit in the large buffer attached to a memory bank). While there has been a plethora of recent works on optimizing row-buffer performance, to our knowledge, there is no study that quantifies the full potential of row-buffer locality and impact of maximizing it on application performance.

Focusing on multithreaded applications, the first contribution of this paper is the definition of a new metric called (memory) row reuse distance (RRD). We show that, while intra-core RRDs are relatively small (increasing the chances for row-buffer hits), inter-core RRDs are quite large (increasing the chances for row-buffer misses). Motivated by this, we propose two schemes that measure the maximum potential benefits that could be obtained from minimizing RRDs, to the extent allowed by program dependencies. Specifically, one of our schemes (Scheme-I) targets only intra-core RRDs, whereas the other one (Scheme-II) aims at reducing both intra-core RRDs and inter-core RRDs. Our experimental evaluations demonstrate that (i) Scheme-I reduces intra-core RRDs but increases inter-core RRDs; (ii) Scheme-II reduces inter-core RRDs significantly while achieving a similar behavior to Scheme-I as far as intra-core RRDs are concerned; (iii) Scheme-I and Scheme-II improve execution times of our applications by 17% and 21%, respectively, on average; and (iv) both our schemes deliver consistently good results under different memory request scheduling policies.

SIGMETRICS'15, June 15–19, 2015, Portland, OR, USA.

DOI: http://dx.doi.org/10.1145/2745844.2745867

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)

General Terms

Performance, Design

Keywords

Multicores, memory scheduling, row-buffer locality, row reuse distance

1. INTRODUCTION

With recent trends towards multicore systems and enormous dataset sizes, the memory system is increasingly becoming a bottleneck [1–5]. Unfortunately, traditional optimization techniques based on on-chip caches will have limited scope in coping with this memory bottleneck, simply because the growth in dataset sizes far exceeds the growth in on-chip cache capacities. Employing smart cache management strategies [6–10] can provide a temporary relief (delta improvement) but certainly not a long-term solution. Instead, to address this growing problem, solutions that consider the *entire path of data accesses* (not just cache performance) should be investigated. One of the components of this path is the last-level cache (LLC) misses (off-chip memory requests).

One of the critical factors that determine the performance of LLC misses in modern main memory systems is *row-buffer locality,* which refers to reusing data from a row-buffer, a buffer that acts as a cache for the most recently accessed memory row, as much as possible. There have been several recent papers [11–15] that proposed hardware-based

techniques to maximize row-buffer locality (row-buffer hits). One popular memory-scheduling (reordering) scheme found in current commercial systems is FR-FCFS [11, 12], which gives priority to requests that target the current memory row in the row-buffer over other requests, including the older ones. More recent scheduling techniques [16–20, 36] try to coordinate memory controllers to achieve better row-buffer performance.

The row-buffer optimization techniques proposed in the literature are mostly ad-hoc architectural schemes that use different heuristics. One of the critical questions in this context is to understand the potential and limits of maximizing row-buffer performance. In other words, *what are the maximum benefits one could expect from an ideal scheme that maximizes row-buffer locality?* Motivated by this observation, this paper makes the following main contributions:

• We define a novel metric called intra-core and inter-core memory *row reuse distance* (RRD) and quantify it using a set of 12 multithreaded benchmarks. This characterization indicates that while intra-core RRDs are generally low, leading to good row-buffer performance (locality), inter-core RRDs are very high.

• We propose an ideal off-chip memory request (LLC miss) scheduling policy oriented towards minimizing only intra-core RRDs. The results collected through our experiments clearly show that while minimizing intra-core RRDs does bring performance improvements (17% execution time reduction on average), there is still scope for further improvement.

• We next study the potential of a strategy that considers *both* intra-core and inter-core RRDs. The results show that such an integrated approach generates a 26% better row-buffer performance (in terms of row-buffer hits), resulting in about 21% improvement (on average) in execution cycles. We also compared our schemes with two previous schemes proposed to improve row-buffer locality. Our evaluation shows that our schemes greatly outperform these previous schemes in reducing the programs' execution time.

Overall, our results motivate considering both intra-core and inter-core RRDs in optimizing for memory system performance. Furthermore, our results clearly show that if off-chip memory requests could be reorganized on the core side, we may not need sophisticated memory request schedulers on the memory controller side. To our knowledge, this is the first paper that formalizes the (memory) row reuse distance concept, and demonstrates the impact of optimizing (minimizing) it on the row-buffer locality and overall application performance.

The remainder of this paper is organized as follows. The next section gives an overview of modern memory systems employed by current multicore/manycore architectures, and explains the concept of row-buffer locality. Section 3 describes our experimental platform as well as application programs. Section 4 defines intra-core and inter-core RRDs and quantifies them in the original application codes. Section 5 gives our two schemes oriented towards minimizing RRDs by performing core-side memory request (LLC miss) scheduling. Section 6 gives detailed evaluations of our schemes. Section 7 discusses related work and Section 8 concludes the paper.

2. DRAM BASICS AND ROW-BUFFER LOCALITY

In this section, we introduce the background on contemporary DRAM memory architectures. We base our introduction on DDR3 SDRAM systems [21, 22]. The discussion below is also applicable to other DRAM memories that employ page-based memory architectures. We also provide an introduction to the state-of-the-art memory request scheduling schemes implemented in hardware.

2.1 SDRAM organization

Figure 1 shows the high-level view of a modern DRAM hierarchy. A DRAM consists of one or more building blocks called dual in-line memory modules (DIMMs). In each DIMM, there are several SDRAM ICs that consist of multiple memory banks. Accesses to different memory banks can be serviced in parallel. The actual elements that store memory data are the DRAM cells, and they are organized as a two-dimensional array in each bank. Read or write operations need to provide a DRAM address that contains bank, row, and column information in order to access the data in the DRAM.

Accesses to the two-dimensional memory cell array take place at the granularity of *rows*. To reduce the delay of accessing the memory array, there is a structure called *row-buffer* that can hold the data of an entire row of the memory array. After the row address is asserted, the contents of the accessed memory row are latched in the row-buffer so that subsequent memory accesses to the same row can be served promptly. Figure 2 illustrates how a memory access is executed. In this example, the memory access needs to read data from bank 1. First, the row address select signal is asserted and the data in the same row across all banks are selected (because a row is the basic accessing unit in the memory array). Following that, the selected row data is latched into the row-buffer. Next, the data can be read from the row-buffer using the bank address and column address. Consecutive memory accesses to data in the row-buffer are called *row-buffer hits*. However, if the successive memory requests access different rows, this will lead to a *row-buffer miss*. In such cases, a new row of memory data needs to be read into the row-buffer. Since memory accesses resulting in row-buffer misses need to access the memory array itself, this leads to longer access latencies. *Row buffer locality* refers to repeated accesses to the contents of a given row when its data is loaded into the row-buffer. Taking advantage of row-buffer locality can improve the performance by eliminating the cycles spent in accessing the memory arrays. A series of memory requests that generate a high row-buffer hit rate is said to have good row-buffer locality.

2.2 Impact of the row-buffer on memory access latencies

Since the row-buffer serves as a cache to the memory bank arrays, latencies of memory accesses are *not* uniform, depending on whether the data is found in the row-buffer or not. The memory access delay can be classified into three categories:

• *Row open with bank hit*: The row-buffer is loaded with data and the accessed data happens to be in the row-buffer. In this case, the memory latency incurred is the minimum

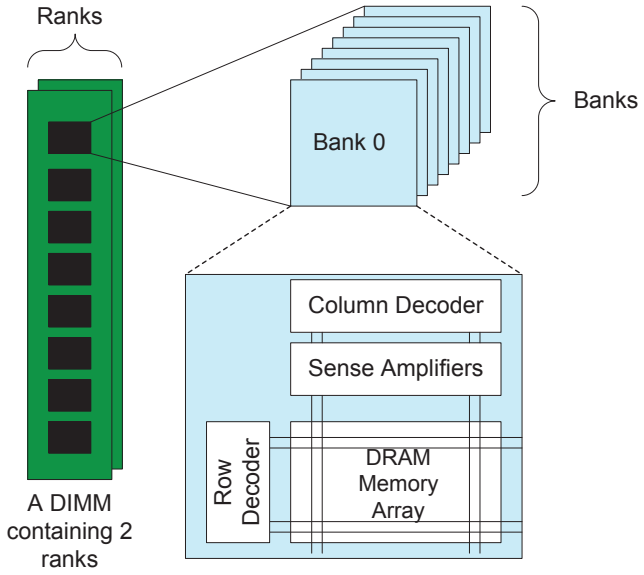


Figure 1: High-level view of a DDR3 SDRAM memory architecture.

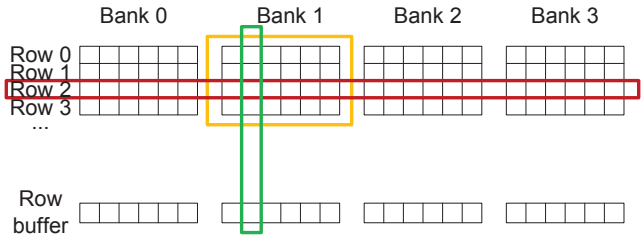


Figure 2: Accessing the memory data through a row-buffer.

and includes only the time to access the right column in the selected bank.

• *Row open with bank conflict*: The row-buffer is loaded with data but the desired data is not in the current row-buffer. The current row needs to be closed first, and the right bank needs to be precharged to load data into the row-buffer. The resulting memory access latency is the highest among all cases.

• *Row empty*: Row buffer is closed in this case and there is no data in the row-buffer. In such a situation, a new row of data need to be loaded into the row-buffer and then desired data can be accessed using column address.

2.3 Row buffer management policies

Existing row-buffer management policies[1] can be categorized into two classes: *open-page policy* and *close-page policy*.

• *Open-page* policy. In this policy, the row-buffer is kept open after every access. This policy is based on the speculation that once a row of data is brought to the row-buffer, the same row may be accessed again in the near future (as a side effect of locality of data). If the next memory access is made to the same row, its memory access latency can be reduced to the minimum since the row is already active and there is no need to access the memory array. However, in the case that the next access is to a different row of the same bank, the memory access latency is much larger, which includes the latency to precharge the DRAM array, access another row and perform the column access. In the open-page policy, the memory access latency is not predictable because a row-buffer hit can have much smaller delay than a row-buffer miss.

• *Close-page policy*. In contrast to the open-page policy, a DRAM page is closed immediately after every read or write operation. This policy does not take advantage of data locality to improve memory bandwidth, but it makes the latency of each memory access predictable and has lower design complexity. The close-page policy can have an edge over the open-page policy in situations where random DRAM pages are accessed frequently.

To summarize, a row-buffer management policy has an influence on the design of the memory system. It directly impacts several other design parameters, such as the memory address mapping and memory access scheduling in the memory controllers. Since our goal in this work is to maximize row-buffer locality, we employ the open-page policy.

[1]Note that these policies just decide whether an accessed row should be kept in the row-buffer for future accesses or not, and are different from memory scheduling policies that reorder requests to improve memory performance.

2.4 Memory access scheduling in the memory controllers

Between processors and DRAM, there is an interface component called *memory controller*. The main functionality of memory controller is to orchestrate the DRAM banks, buses and buffering queues to effectively serve the memory requests from processors. Requests sent from the processors are first stored in the buffering queues. Based on their memory address, the requests are dispatched to the DRAM when there is bus bandwidth available. There is an important logic called *scheduler* (implemented in hardware) in the memory controller that decides the issue order of incoming requests to memory based on performance and priority requirements.

There have been various scheduling schemes proposed in prior work to improve the efficiency and/or fairness of the memory request scheduling. One of the well-known scheduling algorithms is the *First Come First Serve* algorithm (FCFS). In this policy, memory requests are serviced in their arriving order. Note that this policy may not perform well in some cases because it does not take advantage of the data locality in the row-buffer. Instead, the *First Ready–First Come and First Serve* algorithm (FR-FCFS) [11,12] was proposed to optimize the memory throughput. This scheduling policy gives the highest priority to ready memory requests. A memory request is considered ready when its data is available in an open row. If there is no request that leads to a row-buffer hit, the FR-FCFS then selects requests to serve in their arriving order as in the case of FCFS. Serving ready requests first can improve memory throughput by taking advantage of the row-buffer locality and can maximize row-buffer hit rate. The original FR-FCFS scheme was proposed to improve the memory performance (by improving row-buffer locality) in a single-core platform. Recently, there have been several memory scheduling schemes proposed for multi-core platforms to improve performance as well as fairness [20,23–25]. For example, Nesbit et al. [23] propose a scheduling scheme similar to the network fair-queuing strategies to fairly allocate memory bandwidth among applications. Similarly, in [24], a memory scheduling scheme is proposed to

Table 1: Major platform parameters.

Parameter	Default Value
Cores	32 (each is two issue)
Cache Line	64 bytes (for all caches)
L1 Cache	32KB per core (private), 8-way, 4 cycle-latency
L2 Cache	256KB per core (private), 8-way, 12 cycle-latency
L3 Cache	16MB shared, 32-way, 32 cycle-latency (managed as SNUCA [43])
Data TLB	Two-level; L1: 64 entries, 4-way, L2: 512 entries, 4-way
On-Chip Network	4 × 8 mesh, 2-stage wormhole switched, VC flow control,6 VCs per port, 5 flit buffer depth
Main Memory	4 DDR3 Memory Controllers (MC), 8 ranks/MC, 2 banks/rank, FR-FCFS, 2KB row-buffer size, 1107 MHz memory clock, 64GB capacity, 32-entry memory queue size

reorder memory requests based on the average memory stall time of each processor.

3. TARGET SYSTEM, SIMULATION PLATFORM, AND MULTITHREADED APPLICATIONS

Table 1 gives the important features of the default 32-core system we modeled in the Sniper simulator infrastructure [26]. Sniper allows one to carry out timing simulations for both multi-program workloads and multi-threaded, shared-memory applications with tens of cores. Later in the paper, we modify the default values of some of the parameters shown in this table to perform sensitivity experiments.

In this work, we evaluated our approach over a set of 12 multithreaded programs listed in Table 2. The second column gives a brief description of each program, and the third column shows its total input size.[2] The last column on the other hand gives the LLC miss rates of these codes under our default simulation platform described in Table 1. In selecting these applications, we tried to strike a balance between *regular* and *irregular* data access patterns. For example, while benchmarks such as *mgrid* and *fma3d* represent application programs with regular access patterns, *hpcg* and *moldyn* have quite irregular data access patterns made through index arrays. We also tried to have input sizes ranging from relatively small values (210.8 MB) to much larger ones (1.58 GB) to put different amount of pressure on the memory system.

4. ROW REUSE DISTANCE (RRD)

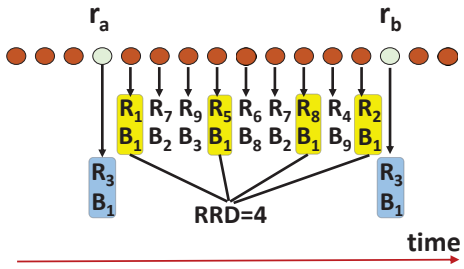
4.1 Definition

The Row Reuse Distance (RRD) between two off-chip accesses (references), r_a and r_b, to the same memory row R_i that reside in memory bank B_j is defined as the number of accesses to different rows $R_k (\neq R_i)$ in the same bank that appear between these two off-chip accesses. For example, in Figure 3(a), the RRD between the two successive accesses to row R_3 (in bank B_1) is 4. Note that we exclude from the reuse distance the references that are targeted to different banks. We observe that RRD can be divided into

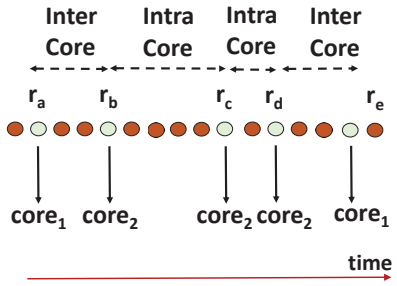
[2]Note that the input sizes we use are larger than the default sizes of these benchmarks to stress the last-level cache (LLC).

Table 2: Applications used in our evaluations.

Application	Brief Description	Input Size	LLC Miss Rate
gs-solver [38]	Gauss-Seidel based iterative sparse solver	390.2MB	22.6%
equake [39]	Earthquake simulation	487.7MB	29.8%
miniFE [40]	Finite element mini application	654.1MB	16.1%
hpcg [42]	High performance preconditioned CG solver benchmark	210.8MB	27.7%
facerec [39]	Face recognition	436.1MB	18.1%
ammp [39]	Chemistry/biology	1.09GB	36.5%
mgrid [39]	Multigrid solver	771.6MB	26.2%
moldyn [41]	Generalized program for the evaluation of molecular dynamics models	336.2MB	21.4%
fma3d [39]	Crash simulation	1.58GB	47.2%
gafort [39]	Genetic algorithm	364MB	20.8%
swim [39]	Shallow water modeling	444.9MB	22.2%
wupwise [39]	Quantum chromodynamics	1.18GB	19.4%



(a)



(b)

Figure 3: (a) Illustration of RRD for row R_3 in bank B_1. r_a and r_b are two requests to R_3. Note that intervening accesses to different rows in the same bank (B_1) contribute to RRD, whereas intervening accesses to different banks do not. (b) Difference between intra-core RRD and inter-core RRD.

two classes: *intra-core RRD* and *inter-core RRD*. As shown in Figure 3(b), if two references belong to the same core, the RRD in question is termed as intra-core RRD. On the other hand, if the references belong to different cores, the corresponding RRD is referred to as inter-core RRD. It is important to emphasize that RRD can be used as a *measure* of the row-buffer locality. More specifically, a small RRD (be it intra-core or inter-core) increases the chances for the second reference (r_b) to generate a row-buffer hit, compared to a large RRD. This is because a small RRD makes

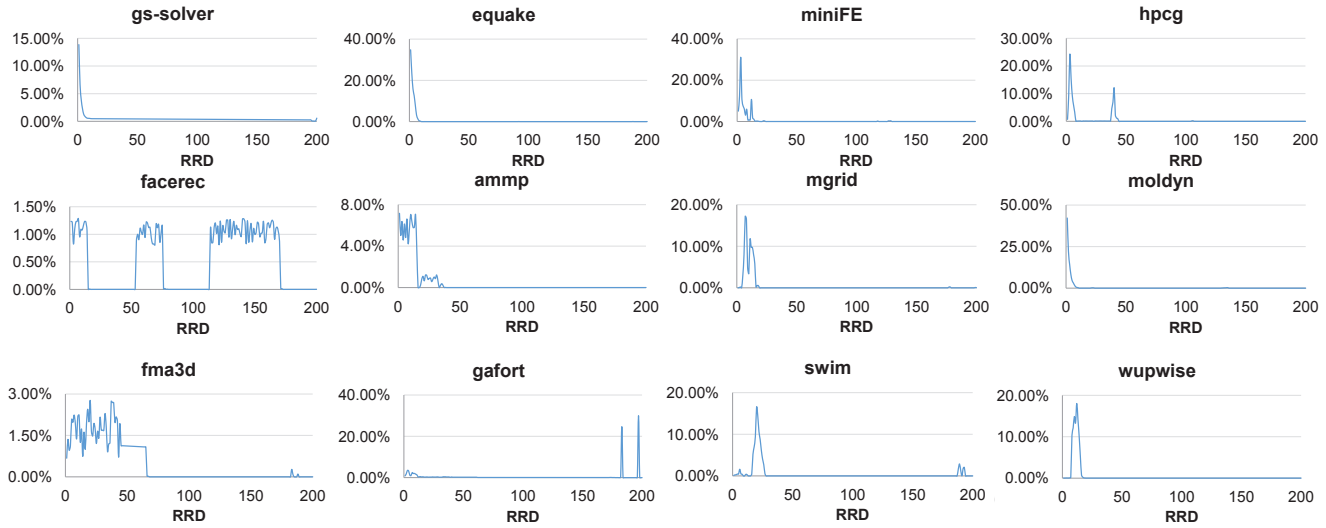


Figure 4: Intra-core RRDs for the original applications. The y-axis in each figure captures the frequency of occurance for different row reuse distances (given on the x-axis). The number of row reuse distances larger than 200 is negligible.

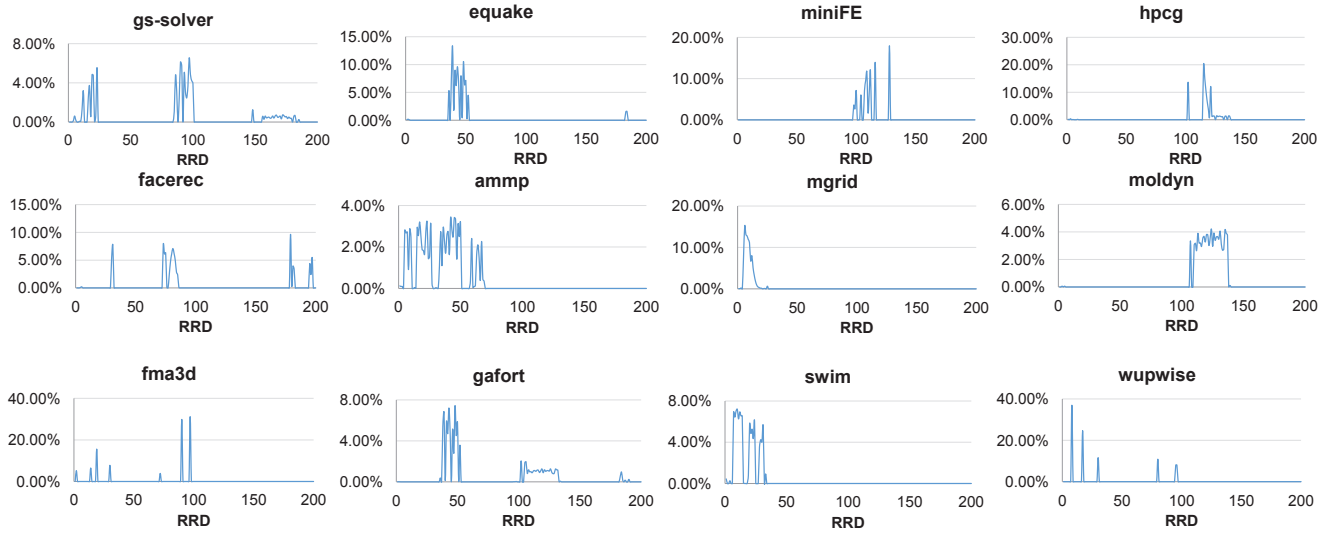


Figure 5: Inter-core RRDs for the original applications. The y-axis in each figure captures the frequency of occurrence for different row reuse distances (given on the x-axis). The number of row reuse distances larger than 200 is negligible.

it more likely for the second reference r_b to be located in the row-buffer, while the first reference (r_a) is in the row-buffer. This scenario results in a row-buffer hit (as opposed to row-buffer miss). Therefore, *one of the optimization strategies that could be built on top of RRDs is to reduce RRDs*, either through architectural techniques, software optimizations, or a combination of both. In the rest of this section, we present intra-core and inter-core RRDs of our multithreaded applications and show how these results translate to row-buffer locality and execution times.

4.2 RRDs of the original applications

Figure 4 and Figure 5 plot, respectively, the distribution of intra-core and inter-core RRDs in our original applications, when considering all rows and all banks. In obtaining the inter-core results, we considered all cores and banks, and similarly, in obtaining the intra-core results (for each core in isolation), all banks are taken into account (note that, in each figure, each point on the x-axis represents a specific RRD value, and the y-axis gives its frequency of occurrence). One can make two observations from these plots. First, intra-core RRDs are generally not very high except for a few applications (e.g., *facerec*, *gafort*, and *fma3d*). It needs to be noted however that only a subset of these row reuses can be converted into row-buffer locality at runtime, depending on the memory buffer size and the memory request scheduling algorithm employed. Specifically, as explained earlier, while FR-FCFS exploits row reuse for the requests waiting in the memory buffer, it cannot exploit any reuse that is not in the buffer. Second, most of the inter-core RRDs plotted in Figure 5 are very high compared to intra-core RRDs. For example, in applications such as *miniFE*, *hpcg* and *moldyn*, almost all RRDs are larger than 100.

While the distribution of RRDs is certainly important, one also needs to consider the contributions (frequency of occurrence) of the intra-core and inter-core row reuses. The plot given in Figure 6(a) shows this breakdown of row reuses between intra-core and inter-core cases. One can observe

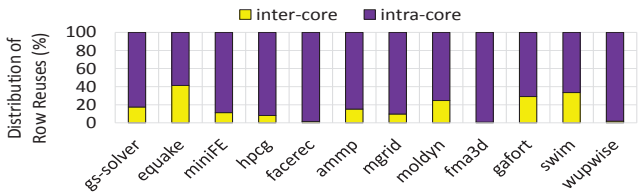
from this plot that, while in some applications such as *facerec* and *wupwise*, intra-core row reuse clearly dominates, in others such as *equake* and *gafort*, inter-core reuse takes a sizable portion. Figure 6(b) gives, under the FR-FCFS scheduling policy and the default values of our system parameters, the row-buffer hit rates of our applications within and across cores. We observe from this plot that the hit rates due to intra-core reuses range between 4% and 42%, averaging on 19%. In contrast, the inter-core hit rates range between 1% and 22%, leading to an average hit rate of 7%. That is, row reuses originated from the same core take much better advantage of row-buffers, compared to reuses across different cores. These hit rates along with the row reuse frequencies plotted in Figure 6(a) contribute to the execution times shown in Figure 6(c).

There have been studies in the past on cache reuse distance. Cache reuse distance [27–29] is a concept proposed to predict the data locality and is employed in techniques to improve cache performance. The cache reuse distance is defined as the number of distinct cache line addresses accessed between two consecutive accesses to the same address. Cache performance can be largely predicted by the reuse distance because it reflects the temporal locality of its data access. For example, in a fully-associative cache employing the LRU (Least Recently Used) replacement policy, we can forecast if a data access will result in a hit or a miss. If the reuse distance of one cache access is greater than the cache size, the next access to the same address will be a miss; otherwise, it will be a cache hit. In reality, cache performance is also affected by other factors such as cache configuration and replacement policies. However, the cache reuse distance can still help to predict the cache performance with certain level of accuracy. In fact, there have been several techniques proposed previously to improve the cache performance based on the reuse distance characteristics [19, 27–30].

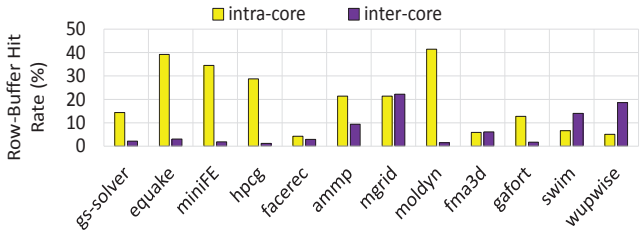
Our proposed RRD and cache reuse distance have significant differences. First, our proposed RRD reflects characteristics of a program's off-chip memory requests, whereas cache reuse distance focuses on on-chip data accesses. Second, our proposed RRD takes into account the specific design of memory banks which is an important factor affecting the performance of memory accesses. For example, intervening accesses to different banks do not contribute to RRD, whereas intervening accesses to different rows of the same bank do. Third, the ways in which cache reuse distances and our RRDs are used for optimization are quite different. Specifically, cache reuse distances are used to improve cache hits, whereas our RRDs target the row-buffer locality of cache misses.

5. MINIMIZING RRDS

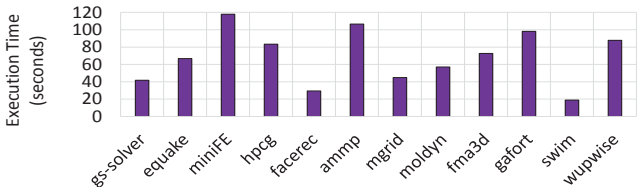
The goal of this section is to present two schemes that aim to reduce RRDs. Both these schemes employ LLC miss reordering (restructuring) to reduce RRDs as much as possible, constrained only by program dependencies. The first of them focuses only on intra-core RRDs, whereas the second one considers both intra-core RRDs and inter-core RRDs. Below, we present the details of these two schemes. It needs to be emphasized that these schemes are in a sense *ideal* approaches, as far as optimizing for intra-core RRDs and intra-core + inter-core RRDs are concerned. Specifically, they do *not* represent any specific implementation; in fact, they cannot directly be implemented in practice. Rather,



(a) Breakdown of row reuses between intra-core and inter-core cases.



(b) Row-buffer hit rates of intra-core and inter-core row reuses.



(c) Execution times of our original applications.

Figure 6: Characterization of row-buffer reuse.

they indicate what would be the results if an ideal cache miss reordering strategy could be employed, constrained by only data and control dependencies in a given program code. In the remainder of this paper, we refer to these two (ideal) schemes as **Scheme-I** and **Scheme-II**.

5.1 Scheme-I: intra-core RRD optimization

Originally, off-chip accesses are sent over to their target memory controllers via the on chip network, and are then forwarded to their target memory banks. Scheme-I reorders those accesses from a single core's view before they reach the memory controllers. Since off-chip accesses are generated strictly following the time-line, to perform our reordering, we construct a very large queue (*buffer*) to temporarily buffer those accesses[3]. Note that initially this queue contains the memory access sequence in its original order. Because different memory banks accommodate disjoint rows, we focus on the row accesses within the same bank by splitting this queue into sub-queues based on the accessed bank's $bank_{id}$. Row buffer locality is then optimized within each of those sub-queues in an isolated fashion. The only factor that prevents Scheme-I from minimizing intra-core RRDs (by reordering the LLC misses in the 128K-entry buffer) is potential data and control dependencies in the program code. Although accesses to different row-buffers will not have memory location dependencies, data hazards from the

[3]In our experiments, we used a 128K-entry buffer. Increasing the size of this buffer further did not bring any additional benefits when all the requests in this buffer have been processed, we reload it with the next 128K LLC misses.

program will restrict the opportunities for maximizing row-buffer locality. Consider the following scenario as an example. In a three-instruction case, an arithmetic instruction first generates an access to row_2 due to an LLC miss. A load instruction then loads a value from row_1 to register t_1. Finally, a store instruction stores t_1 to row_2. In this scenario, one cannot swap the load and store instruction to achieve row_2 locality due to the load-store hazard. In general, any dependences between the instructions that access the same row can prevent Scheme-I from minimizing RRDs. Note that, in a practical implementation, memory dependencies can be handled in different layers, and we do not defend one possible implementation over the other. Rather, our goal is to take into account the impact of dependencies in an otherwise ideal (unimplementable) scheme that can perform core-side memory scheduling considering a very large buffer of references.

Our miss-reordering based approach to intra-core RRD optimization is given as a pseudo-code in Algorithm 1. In this pseudo-code, the body of the outermost loop is assumed to be executed for all cores in the system. X and Y are the number of channels and the number of banks per channel, respectively. Each bank has a link list as its queue, which is initialized in *line 27*. Our approach then distributes the accesses over the access sub-queues according to each access's $bank_{id}$, which is implemented by a loop between *line 28* and *line 30*. We then call the reorder function to optimize row-buffer locality. The first loop (*line 2*) in this function traverses all memory bank sub-queues, while the second loop (*line 4*) traverses all accesses within each queue. The while loop (*line 10*) is used to find accesses to the same row. Note that, in *line 12*, if a dependency is detected or the *RRD* is already very small (which can be captured by FR-FCFS anyway), we skip this access and go to the next one.

We now go over a simple example, shown in Figure 7, to illustrate how our scheme works in practice. Figure 7(a) depicts the original access sequence to two memory banks. All accesses are originally ordered following the time-line without considering row-buffer locality (representing the original LLC miss sequence). Figure 7(b) shows the sub-queues according to $bank_{id}$. Dependencies are marked by an arrow, and Figure 7(c) illustrates the reordered bank queues after applying our strategy. Originally, we have 6 row-buffer misses to $bank_1$ and 6 row-buffer misses to $bank_2$ in Figure 7(b). We reduce the row-buffer misses to 3 in $bank_1$ and 4 in $bank_2$, respectively. Note that, although the best order will give us 2 row misses to $bank_1$ because of the three row accesses in total, we cannot achieve that due to dependencies.

So far, we ordered the LLC misses from each core's perspective independently. It needs to be noted that, evaluating Scheme-I is important as it is relatively easier to incarnate a practical implementation from it, compared to Scheme-II (presented shortly). This is because the former does not require any inter-core coordination which would be very challenging to implement in practice without significant hardware complexity and runtime overheads. Also note that, although memory references from different cores interleave at memory controllers and banks, maintaining core-level locality of memory accesses is still expected to achieve some locality improvement (especially in cases where a given bank is shared – in a given period – by only few cores). However, Scheme-I may not be able to provide the optimal row locality, when accesses from all cores are considered together. Suppose, for example that, $core_1$ accesses row_1 and row_2, and $core_2$ accesses the row_3 and row_1 from the same memory bank after $core_1$ accesses row_1. In this case, the single core view adopted by Scheme-I will not be sufficient to optimize the locality of row_1, which leads us to our intra-core + inter-core RRD optimization.

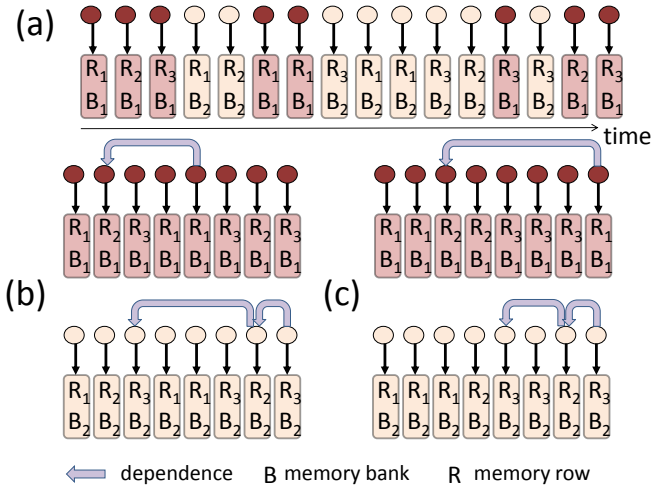


Figure 7: Example for intra-core RRD optimization.

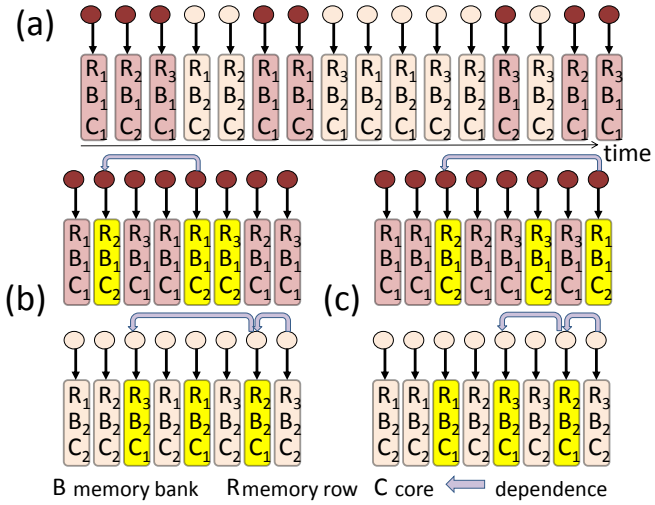


Figure 8: Example for inter-core RRD optimization.

5.2 Scheme-II: intra-core + inter-core RRD optimization

We now discuss an alternate scheme, which considers *both* intra-core row reuse and inter-core row reuse. Instead of focusing on each core in isolation, Scheme-II expands the scope to all the cores in the system, that is, it looks at the LLC misses coming from all cores at a given period of time (within a queue length of 128K entries). Recall that, in the scenario we discussed at the end of the previous section, $core_1$ accesses row_1 and then $core_2$ accesses row_3 and row_1. In this scenario, the two accesses from $core_2$ can be swapped to enable the row reuse with $core_1$. A pseudo-code version of this approach is provided in Algorithm 2.

Algorithm 1 Scheme-I

```
INPUT: # of Channels: X, # of Banks/channel: Y, Mem channel
    queue length: N, # of cores: K
 1: function ACCESSES-REORDER(X,Y)
 2:     for bank_j, j from 0 to X*Y do
 3:         num ← access misses in bank_j
 4:         for miss_k, k from 0 to num do
 5:             miss_var ← miss_k
 6:             if miss_var is marked as scanned then
 7:                 continue
 8:             end if
 9:             scan bank_j
10:             while next miss_l accesses the same row-buffer do
11:                 RRD ← distance between these two misses
12:                 if dependence_detected || RRD ≤ N then
13:                     miss_var ← miss_l
14:                 else
15:                     move miss_l next to miss_k
16:                     update bank_j
17:                     miss_var ← miss_l
18:                 end if
19:                 mark miss_l scanned
20:             end while
21:         end for
22:     end for
23: end function
24: // outermost loop
25: for core C_i, i from 0 to K do
26:     //initial all bank queue sets
27:     bank_j ← ∅   j from 0 to X * Y
28:     for each access miss do
29:         bank_{bank_id} ← access_{bank_id}
30:     end for
31:     //reorder access misses
32:     ACCESSES-REORDER(X,Y)
33: end for
```

Algorithm 2 Scheme-II

```
INPUT: # of Channels: X, # of Banks/channel: Y, Mem channel
    queue length: N, # of cores: K
1: //initial all bank queue sets
2: bank_j ← ∅   j from 0 to X * Y
3: for core C_i, i from 0 to K do
4:     for each access miss do
5:         bank_{bank_id} ← access_{bank_id}
6:     end for
7: end for
8: //reorder access misses
9: ACCESSE-REORDER(X,Y)
```

Instead of generating a separate bank queue for each core, we gather the accesses from all the cores and store them in a set of very large queues based on access arrival time and relative $bank_{id}$, (captured between *line 3* and *line 7*). We then apply our miss reordering strategy on the bank queues which now contain mixed accesses from all the cores.

Figure 8 shows an example illustrating how Scheme-II works in practice. In this example, the original queue contains accesses from two cores, as illustrated in Figure 8(a). Similar to the intra-core classification method, we build the queues according to access $bank_{id}$ in Figure 8(b). Figure 8(c) depicts the optimized access order. Note that without considering inter-core reuse, we would have an access order of $R_2R_1R_3$ for memory $bank_1$ of $core_2$. However, taking into account inter-core reuse, Scheme-II achieves an access order $R_2R_3R_1$, which benefits the row-buffer locality between $core_1$ and $core_2$ on accessing R_3.

It is to be noted that, as far as intra-core reuse is concerned, Scheme-II is expected to be almost as good as Scheme-I. This is because the main difference between these two schemes is that, in Scheme-II we mix all the core accesses together. Note also that the bank queue generation depends only on $bank_{id}$. Our reorder function only takes $bank_{id}$ and row_{id} as input parameters, which are orthogonal to $core_{id}$. As a result, all intra- and inter-core row reuses are captured.

5.3 Discussion

We want to emphasize that neither Scheme-I nor Scheme-II can be implemented in hardware as they are. They are meant to be used as bars against which results of practical implementations can be compared. In this subsection, we would like to discuss the hurdles that need to be addressed if one wants to come close to the performance of our ideal schemes. Let us start with Scheme-I first. A practical implementation that approximates Scheme-I should be able to (i) reorder off-chip references (LLC misses) in a very large buffer before sending them over to their respective target banks, and (ii) ensure that these requests are not reordered in the on-chip network. One difficulty in achieving (i) is the fact that one may not be able to afford to accommodate a very large buffer (e.g., 128K entries) on the core side. Another difficulty is that the potential dependences between off-chip requests need to be checked at the runtime, and costs of such checks usually increase with increasing buffer size. The issues here would be very similar to those encountered in designing re-order buffers (ROB [44]) in superscalar processor design. Point (ii) is even more problematic, because different requests can be delayed in on-chip network (between cores and memory controllers) by different amounts, thereby arriving controllers in a different order than they exit cores. Ensuring that the off-chip requests are not reordered in the network may not be trivial in practice. However, an alternate view of this problem could be to let the network routers themselves implement the desired ordering of off-chip memory requests. That is, each router can be attached with a reasonable-sized buffer using which the off-chip requests are ordered. If all routers implement this policy, one can potentially expect very good row-buffer performance on the DRAM side. Instead of modifying all network routers, one may also consider modifying only the ones that are close to the memory controllers. Coming to Scheme-II, it is certainly much more difficult to implement it in practice, because it requires a global coordination across all cores to decide on the best order of off-chip requests. Note however that the network router-based implementation alternative mentioned above can achieve, to some extent, the effect of Scheme-II, as each on-chip network router normally handles requests coming from different cores and destined to different channels/banks. In case one prefers to have a core-side implementation however (inside of a router-based one), software (e.g., OS) based implementation can read buffers attached to cores and reorder them from a global perspective. Clearly, minimizing the runtime overheads in this case will be a primary concern.

6. EXPERIMENTAL EVALUATION

6.1 Results with the default system parameters

We now quantify the impact and benefits brought by the two schemes discussed in the previous section when simulating a system whose major parameters are listed in Table 1. First, we present in Figure 9 and 10 the distribution of intra-core and inter-core RRDs, respectively, when Scheme-I is applied. One can observe from these results that, while, as

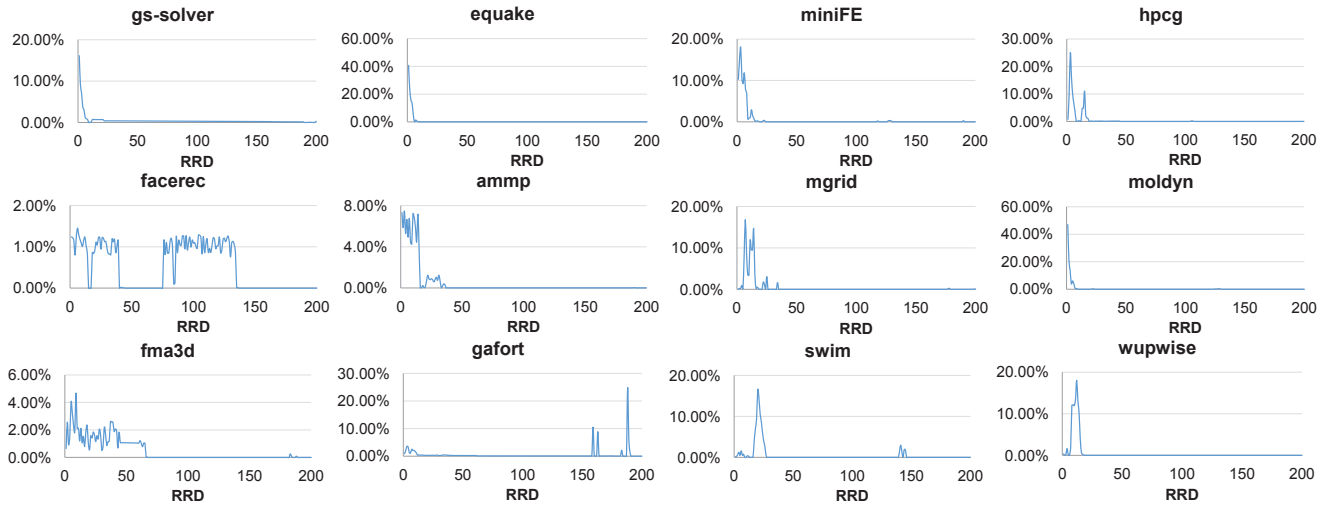


Figure 9: Intra-core RRDs with Scheme-I. The y-axis in each figure captures the frequency of occurrence for different row reuse distances (given the on x-axis). The number of row reuse distances larger than 200 is negligible.

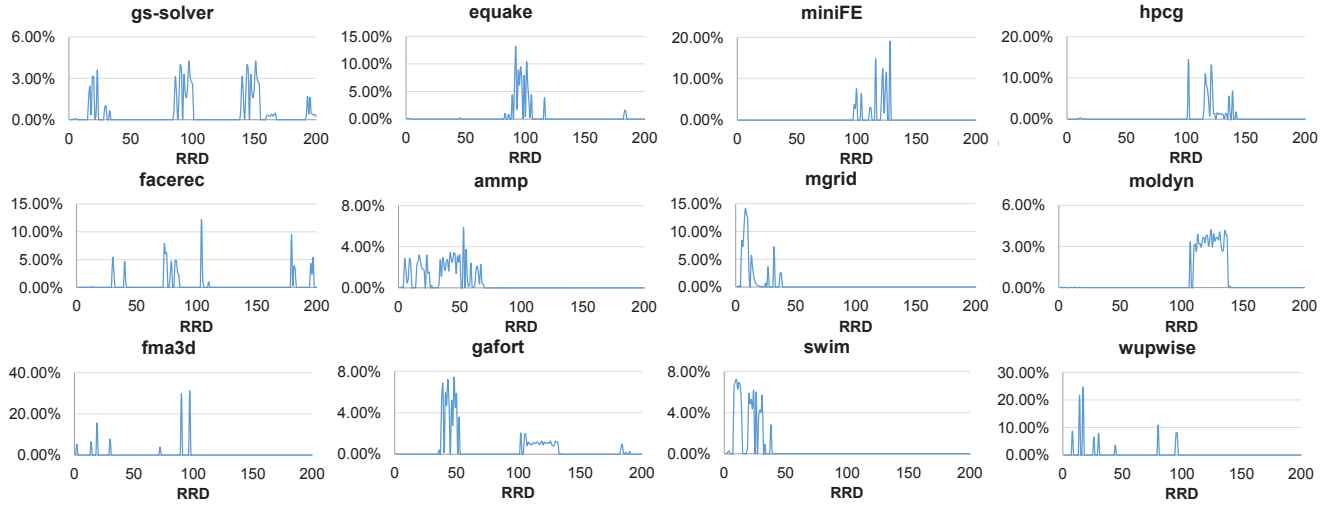


Figure 10: Inter-core RRDs with Scheme-I. The y-axis in each figure captures the frequency of occurrence for different row reuse distances (given on the x-axis). The number of row reuse distances larger than 200 is negligible.

expected, Scheme-I significantly reduces intra-core RRDs, it (unexpectedly) increases inter-core RRDs. This result means that optimizing row reuse from the perspective of each core in isolation does not necessarily generate good row-buffer locality from a global perspective. The first two groups of bars in Figure 13 give the row-buffer hit rates (higher is better) with this scheme. Comparing these results to those presented earlier in Figure 6 show that Scheme-I improves the intra-core row-buffer locality by 16% on an average. In turn, the impact of these row-buffer hit rate improvements on execution times are plotted in Figure 14 (the first bar for each benchmark), as *percentage improvements over the original case*. Overall, optimizing for only intra-core RRD results in an execution time reduction of 17%, when averaged over all 12 application programs we used.

Figures 11 and 12, on the other hand, give the distribution of intra-core and inter-core RRD distributions with Scheme-II. One can make two important observations from these results. First, the distribution of intra-core misses in Figure 11 is very similar to that in Figure 9. That is, considering inter-core row reuse does not significantly impact intra-core reuse distances. The second observation is that Scheme-II substantially reduces the inter-core RRDs. These results in turn translate to the row-buffer hit rates plotted in Figure 13 (the last two groups of bars), indicating an average row-buffer hit rate improvement of 12% for intra-core hits and 13% for inter-core hits. Finally, this scheme brings execution time improvements ranging between 7% (*facerec*) and 34% (*ammp*), as captured by the second bar for each benchmark in Figure 14.

6.2 Evaluation of the previously-proposed schemes

In this subsection, we report results from our implementation of two previously-proposed schemes that improve row-buffer locality and show how close they come to our savings reported above. This first of these schemes, by Awasthi et al. [18], employs adaptive first-touch page-placement and dynamic page-migration, and the execution time improvements it generates over the original execution case are given as the third bar for each application in Figure 14. It can be seen that this approach reduces execution time by 10%

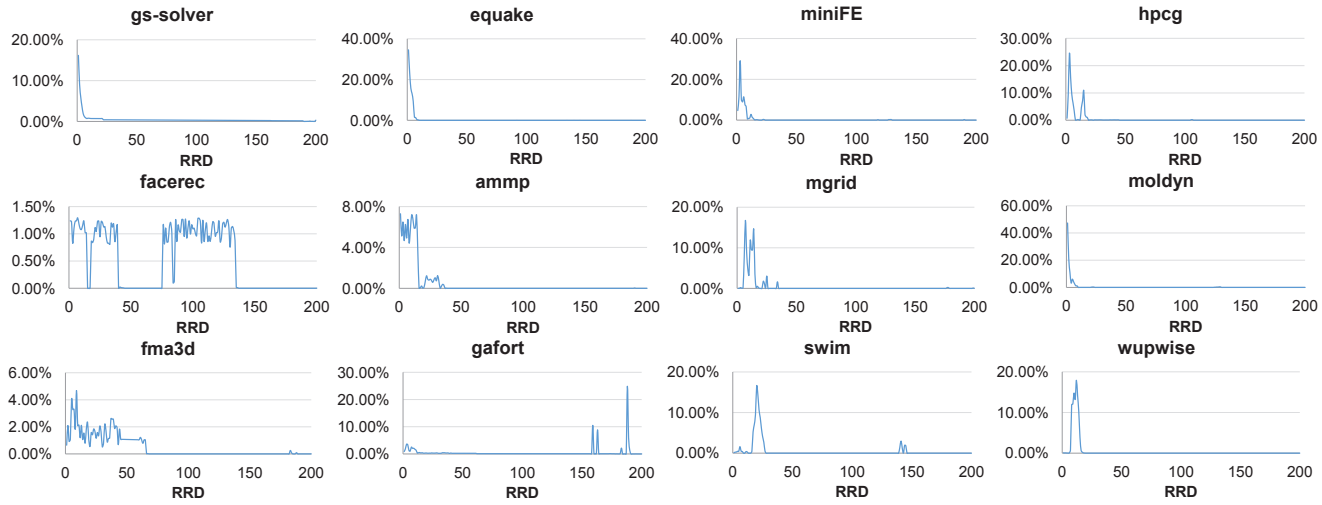


Figure 11: Intra-core RRDs with Scheme-II. The y-axis in each figure captures the frequency of occurrence for different row reuse distances (given on the x-axis). The number of row reuse distances larger than 200 is negligible.

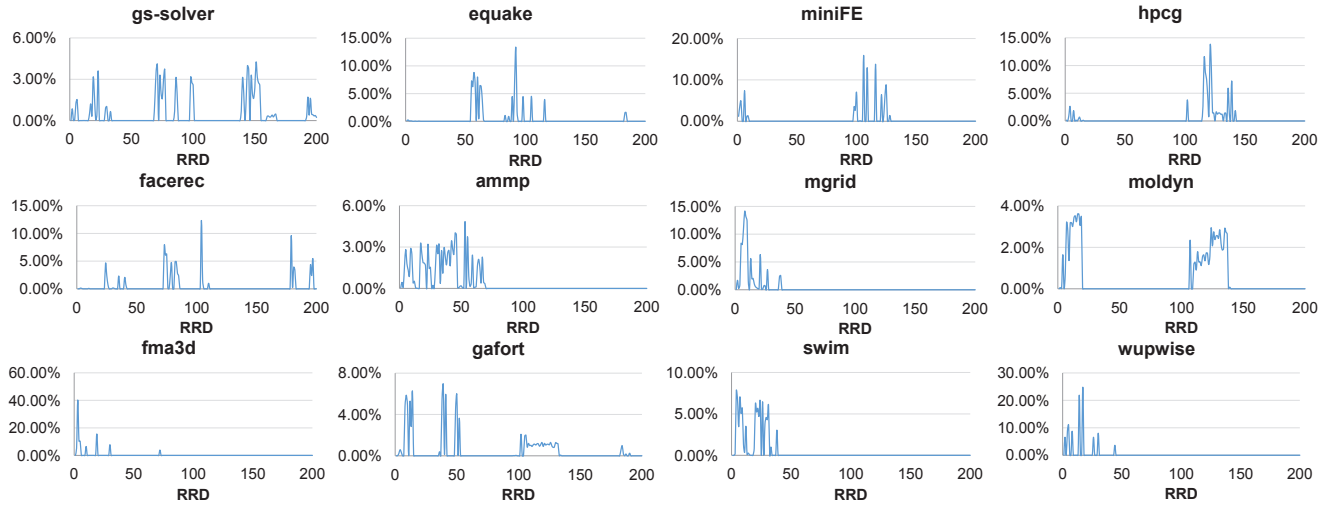


Figure 12: Inter-core RRDs with Scheme-II. The y-axis in each figure captures the frequency of occurance for different row reuse distances (given on the x-axis). The number of row reuse distances larger than 200 is negligible.

on an average. The second scheme we tested [37] is not specific to multithreaded applications but can be used for them. It orchestrates page frame allocation so that the pages that threads access are dispersed randomly across multiple banks so that each thread's access pattern is randomized. The execution time improvements it brings over the original case are plotted as the fourth bar in the same figure (9% average improvement). These results clearly show that, while both these practical schemes are effective in improving performance, there is a significant gap between them and our ideal schemes, motivating for further research on improving row-buffer locality.

6.3 Results from the sensitivity experiments

Our goal in this section is to evaluate our schemes under different values of our major system parameters. In each experiment, the value of only one parameter is modified; the remaining parameters retain their original values listed in Table 1. The results from these sensitivity experiments with Scheme-II are given in Figure 16. Our main observations from these experiments can be summarized as follows. First, when we increase the number of cores, the effectiveness of our scheme increases. This is mainly because a higher core count makes considering the row reuse across cores even more critical, increasing opportunities for Scheme-II. A similar observation, with a smaller variance though, can be made with the increased LLC (L3) capacity. This is due to the fact that a larger L3 causes the off-chip requests to spread apart more and become sparser over the memory space, this again benefits more from our scheme, which is oriented towards reducing large reuse distances. However, an opposite trend is observed when the row-buffer (memory row) size increased. This is because a larger memory row makes some of the memory accesses that would normally generate row-buffer misses result in row-buffer hits. Next, increasing the number of banks (by keeping the total memory space same) causes the off-chip accesses to spread more over the address space, again increasing the importance of row-buffer optimization. Finally, as expected, reducing the memory queue size helps our schemes generate better results, as a large memory queue captures some of the reuse that would otherwise be lost in the smaller queue. Overall, these sensitivity experiments clearly show that there is a great potential for

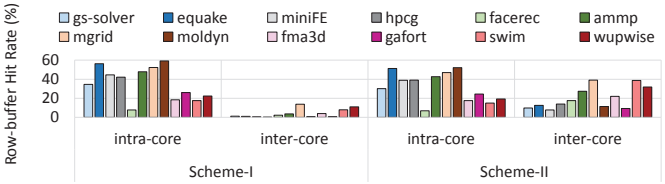


Figure 13: Row-buffer hit rates with our schemes.

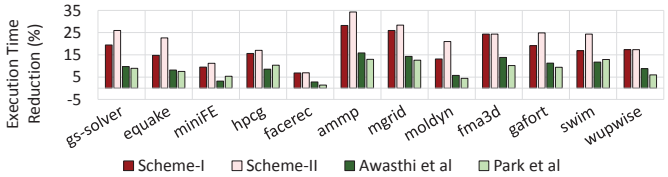


Figure 14: Execution time improvements with different schemes.

Scheme-II under various values of our major system parameters.

6.4 Results with different memory scheduling policies

Our final set of experiments measure the behavior of our two schemes under different memory request scheduling policies. Recall that the default memory scheduling policy used in our experiments so far was FR-FCFS, which re-orders the pending requests in the memory buffer to maximize row-buffer hits. Since our schemes also reorder the requests (on the core side) by looking at a much larger window (128K entries instead of 32K entries), it is interesting to test its impact under different memory scheduling policies. In the results presented in Figure 15, we use our approach with FCFS (simple first-come, first-served; which is basically FR-FCFS with no locality engagement), ATLAS [20], and TCM [35]. ATLAS [20] periodically orders threads based on the service they received from the memory controllers so far, and prioritizes the threads that have attained least service over others. In comparison, TCM [35] groups threads with similar memory access behavior into either the memory-non-intensive or the memory-intensive bin, and performs prioritization across these bins to maximize memory performance. In Figure 15, all bars for a given benchmark are *normalized* with respect to our approach (Scheme-II) running with FR-FCFS. Maybe the most striking observation from this plot is that these different memory scheduling policies generate very similar performance results (between -3% and 3% range), as long as they are used with our scheme. In other words, *the existence of our scheme makes the specifics of the underlying memory scheduling policy much less important.* This is a very important (and somewhat unexpected) result because, everything else being equal, it is always more cost efficient to employ the least complex policy at hardware. These results clearly show that, if we could reorder the off-chip requests on the core side, there is *no* need for any second-level request reordering (in hardware) at the memory bank queues.

7. RELATED WORK

Reuse distance has been studied in a lot of prior works to help design techniques that can improve the program's

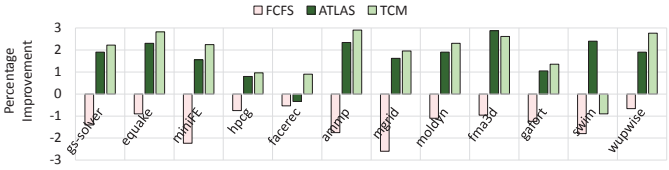


Figure 15: Results with different memory scheduling policies. For each benchmark, all bars are normalized to Scheme-II running with FR-FCFS.

performance [27–29, 31–34]. Chen and Zhong proposed a scheme to predict program locality by analyzing the data reuse distance with the help of profiling techniques [27]. They first employed distance-based sampling to a program's execution. Then they use a pattern recognition technique to analyze the data reuse distance to predict the program's data locality. Keramidas et al used reuse distance to improve the traditional LRU cache replacement policy [28]. Based on their observation that LRU is not effective for applications that have a large reuse distance, they proposed static and dynamic cache replacement policies in order to improve performance.

Zhang et al [19] investigated on multi-core data reuse. Their investigation shows that current on-chip cache hierarchies of multicore architectures and the state-of-the-art code/data optimizations are not able to fully exploit inter-core data reuse in parallel applications. They presented a compiler-based data locality optimization strategy that can balance both inter-core and intra-core reuse optimizations. Our work is different from [19] because we focus on row-buffer data locality.

There have also been many previously-proposed schemes to optimize the design of memory schedulers [13, 14, 18, 20, 23–25]. Zhang et al proposed a permutation-based scheme that dramatically increases the row-buffer hit rate to reduce memory stall time [13]. Their scheme employed the fast exclusive-OR operation to generate the bank index, so that data addresses are interleaved in a way to increase both the bank-level parallelism as well as data locality in the row-buffer. However, their scheme targets only on single processor platforms. In this work, we are more interested in measuring the impact of row-buffer locality on performance for multithreaded applications.

The past years have also seen many research works leveraging memory controller scheduling to improve the fairness for parallel applications [20, 23–25, 35, 36]. A scheme is proposed in [20] that periodically reorders threads based on the amount of service they have obtained from the memory controllers. Their scheme is based on Pareto Distribution which claims that a job running for a short time so far is expected to end soon. The proposed memory scheduling policy prioritizes threads receiving the least amount of service over others in each epoch in order to provide fairness to all threads. PAR-BS [25] is a memory scheduling scheme proposed to provide quality of service and also improve the system throughput. In the PAR-BS scheme, DRAM requests are processed in batches in order to provide fairness. PAR-BS also employs parallelism-aware DRAM scheduling by processing requests from a given thread in parallel across all the DRAM banks. As a result, the system performance

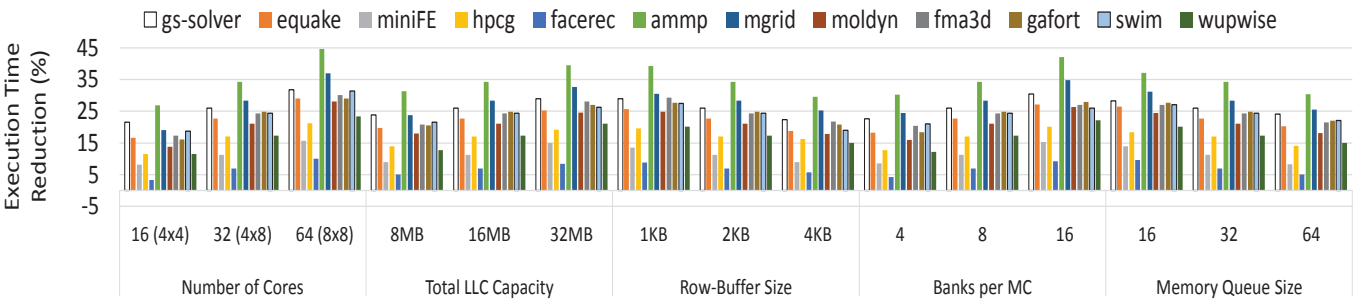


Figure 16: Results from the sensitivity experiments. In each sensitivity experiment, the middle group of bars represent the default value.

can be improved by reducing the memory stall time of the threads.

8. CONCLUDING REMARKS AND FUTURE WORK

In this paper, we make two main contributions. First, we define row reuse distance (RRD) both for a core and across different cores, and quantify it using a set of 12 multithreaded benchmarks and a detailed simulation infrastructure. Second, we evaluate two ideal schemes targeting (i) only intra-core row reuses and (ii) both intra- and inter-core memory row reuses. The results from this evaluation show that (i) the scheme that targets only intra-core row reuses results in an increase in inter-core reuse distances (despite the significant reduction in intra-core reuse distances), and (ii) the second scheme significantly improves inter-core row reuse distances while maintaining almost the same performance level of the first scheme regarding the intra-core row reuses. To our knowledge, ours is the first work that formalizes the concept of memory row reuse distance concept, and gives two schemes to optimize it. Overall, our experimental result indicate that the proposed two schemes can improve row-buffer buffer hit rates by 13% and 26% on average, which translate to average execution time improvements of 17% and 21%. The schemes evaluated in this work are not directly implementable. Our next step includes developing practical schemes that can approximate the behavior/performance of these ideal schemes. In particular, motivated by the observation that considering both intra-core and inter-core row reuse distances can bring significant execution time savings, we plan to explore low-overhead techniques that enable coordinated LLC miss scheduling across multiple cores.

9. ACKNOWLEDGMENTS

This work is supported in part by NSF grants 1213052, 1205618, 1439021, 0963839, and 1017882, as well as a grant from Intel.

10. REFERENCES

[1] M. Xie, D. Tong, K. Huang and X. Cheng, *Improving system throughput and fairness simultaneously in shared memory CMP systems via Dynamic Bank Partitioning*, HPCA, 2014.

[2] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, *The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost*, ICCD, 2014.

[3] B. T. Davis, *Modern DRAM Architectures. PhD thesis*, University of Michigan, 2000.

[4] W. Ding, D. Guttman and M. Kandemir, *Compiler Support for Optimizing Memory Bank-Level Parallelism*, MICRO, 2014.

[5] S. O,Y. H. Son, N. S. Kim and J. H. Ahn, *Row-buffer decoupling: a case for low-latency DRAM microarchitecture*, ISCA, 2014.

[6] D. Chandra, F. Guo, S. Kim, and Y. Solihin. *Predicting inter-thread cache contention on a chip multi-processor architecture*, HPCA, 2005.

[7] J. Chang and G. S. Sohi, *Cooperative cache partitioning for chip multiprocessors*, ICS, 2007.

[8] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely Jr. and J. Emer, *Adaptive insertion policies for managing shared caches*, PACT, 2008.

[9] M. Kandemir, S. P. Muralidhara, S. H. K. Narayanan, Y. Zhang, O. Ozturk, *Optimizing shared cache behavior of chip multiprocessors*, MICRO, 2009.

[10] S. Kim, D. Chandra and Y. Solihin *Fair cache sharing and partitioning in a chip multiprocessor architecture*, PACT, 2004.

[11] S. Rixner, *Memory controller optimizations for web servers*, MICRO, 2004.

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

[13] Z. Zhang, Z. Zhu, and X. Zhang, *A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality*, MICRO, 2000.

[14] S. M. Zahedi, and B. C. Lee, *REF: resource elasticity fairness with sharing incentives for multiprocessors*, ASPLOS, 2014.

[15] H. Wang, R. Singh, M. J. Schulte, and N. S. Kim, *Memory scheduling towards high-throughput cooperative heterogeneous computing*, PACT, 2014.

[16] J. Hasan, S. Chandra, and T. N. Vijaykumar, *Efficient Use of Memory Bandwidth to Improve Network Processor Throughput*, ISCA, 2003.

[17] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding and O. Mutlu, *Row Buffer Locality Aware Caching Policies for Hybrid Memories*, ICCD, 2012.

[18] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian and A. Davis, *Micro-Pages: Increasing DRAM Efficiency with Locality-Aware Data Placement*, ASPLOS, 2010.

[19] Y. Zhang, M. T. Kandemir and T. Yemliha, *Studying inter-core data reuse in multicores*, SIGMETRICS, 2011.

[20] Y. Kim, D. Han, O. Mutlu and M. Harchol-Balter, *ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers*, HPCA, 2010.

[21] JEDEC Solid State Technology Association, *DDR3 SDRAM Specification*, JESD79-3D edition, Sept, 2009

[22] *Calculating Memory System Power for DDR3*, Technical report, Micron Technology Inc., 2–7, TN-4-01, 2007.

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

[24] O. Mutlu and T. Moscibroda. *Stall-time fair memory access scheduling for chip multiprocessor*, MICRO, 2007.

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

[26] T. E. Carlson, W. Heirman, and L. Eeckhout, *Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations*, SC, 2011.

[27] D. Chen and Y. Zhong, *Predicting whole-program locality through reuse distance analysis*, PLDI, 2003.

[28] G. Keramidas, P. Petoumenos and S. Kaxiras, *Cache Replacement Based on Reuse-Distance Prediction*, ICCD, 2007.

[29] A. Jaleel, K. B. Theobald, S. C. Steely Jr. and J. Emer, *High Performance Cache Replacement Using Re-Reference Interval Prediction*, ISCA, 2007.

[30] K. Beyls and E. H. D'Hollander, *Reuse distance as a metric for cache behavior*, IPDCS, 2001.

[31] G. Almasi, C. Cascaval and D. A. Padua, *Calculating stack distances efficiently*, SIGPLAN Not., 2003

[32] Y. Jiang, E. Z. Zhang, K. Tian, X. Shen, *Is reuse distance applicable to data locality analysis on chip multiprocessors?*, Compiler Construction, 2010.

[33] M. Kandemir, *A compiler technique for improving whole-program locality*, POPL, 2001.

[34] D. L. Schuff, M. Kulkarni, and V. S. Pai, *Accelerating multicore reuse distance analysis with sampling and parallelization*, PACT, 2010.

[35] Y. Kim, M. Papamichael, O. Mutlu and M. Harchol-Balter, *Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior*, MICRO, 2010.

[36] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian and A. Davis, *Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers*, PACT, 2010.

[37] H. Park, S. Baek, J. Choi, D. Lee and S. Noh, *Regularities considered harmful: forcing randomness to memory accesses to reduce row-buffer conflicts for multi-core, multi-bank systems*, ASPLOS, 2013.

[38] R. Barrett, R. Barrett, M. Berry3, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd Edition, SIAM, 1994.

[39] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady, *SPEComp: A new benchmark suite for measuring parallel computer performance*, WOMPEI, 2001.

[40] https://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks.

[41] D J. Craik, A .Kumar, G. C. Levy, *MOLDYN: a generalized program for the evaluation of molecular dynamics models using nuclear magnetic resonance spin-relaxation data*, J. Chem. Inf. Comput. Sci., 1983.

[42] https://software.sandia.gov/hpcg/html/index.html.

[43] C. Kim, D. Burger, and S. Keckler, *An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches*, ASPLOS, 2002.

[44] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, 4th Edition*, Morgan Kaufmann, 2012.