Trading Cache Hit Rate for Memory Performance

Wei Ding, Mahmut Kandemir, Diana Guttman, Adwait Jog, Chita R. Das, Praveen Yedlapalli
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, Pennsylvania, USA
{wzd109, kandemir, drg217, adwait, das, praveen}@cse.psu.edu

ABSTRACT

Most of the prior compiler based data locality optimization works target exclusively cache locality optimization, and row-buffer locality in DRAM banks received much less attention. In particular, to the best of our knowledge, there is no single compiler based approach that can improve row-buffer locality in executing irregular applications. This presents a critical problem considering the fact that executing irregular applications in a power and performance efficient manner will be a key requirement to extract maximum benefits from emerging multicore machines and exascale systems. Motivated by these observations, this paper makes the following contributions. First, it presents a compiler-runtime cooperative data layout optimization approach that takes as input an irregular program that has already been optimized for cache locality and generates an output code with the same cache performance but better row-buffer locality (lower number of row-buffer misses). Second, it discusses a more aggressive strategy that sacrifices some cache performance in order to further improve row-buffer performance (i.e., it trades cache performance for memory system performance). The ultimate goal of this strategy is to find the right tradeoff point between cache performance and row-buffer performance so that the overall application performance is improved. Third, the paper performs a detailed evaluation of these two approaches using both an AMD Opteron based multicore system and a multicore simulator. The experimental results, collected using five real-world irregular applications, show that (i) conventional cache optimizations do not improve row-buffer locality significantly; (ii) our first approach achieves about 9.8% execution time improvement by keeping the number of cache misses the same as a cache-optimized code but reducing the number of row-buffer misses; and (iii) our second approach achieves even higher execution time improvements (13.8% on average) by sacrificing cache performance for additional memory performance.

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.

http://dx.doi.org/10.1145/2628071.2628082.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*

Keywords

Data locality; compiler; irregular application; cache; row buffer

1. INTRODUCTION

Many data intensive applications are irregular in their data access patterns, inter-thread communication, control flow and storage accesses. Unfortunately, compared to the vast literature we have for dense computations, irregular computations received much less attention from academia and industry. This is partly because optimizing irregular applications is much more challenging than optimizing dense computations as their behavior is significantly more complex and does not lend itself well to static analysis. This presents a critical problem considering the fact that executing irregular applications in a power and performance efficient manner will be a key requirement to extract maximum benefits from emerging multicore machines and future exascale systems.

The most active research areas in the context of irregular applications have been parallelism optimization (i.e., minimizing inter-thread communication [22, 31, 11]) and data locality optimization [13, 15]. It is interesting to note that all prior data locality optimizations targeting irregular programs considered exclusively *cache locality*. However, in modern multicore systems, cache locality is not the only type of locality that matters. For example, current commercial multicores do not only move memory controller on-chip but a majority of them also support multiple memory controllers. Each of these controllers orchestrates data flow into/from multiple memory banks. Row-buffer is a buffer in front of a bank (in DRAM) that holds the most recently-accessed memory row. To keep costs low, the buffering circuitry in DRAM devices is amortized among large rows of cells, resulting in the employment of large (2KB-8KB) buffers in current systems.

While one may think of a row-buffer as another level in the cache hierarchy, it requires special attention because of the following reasons. First, the row-buffer is a single buffer whose "entire contents" are replaced in a row-buffer miss. Therefore, even one intervening access to another row in the same bank can destroy the whole row-buffer locality. Second, a row-buffer serves only for memory accesses issued to the bank it is attached to. This needs to be kept in mind

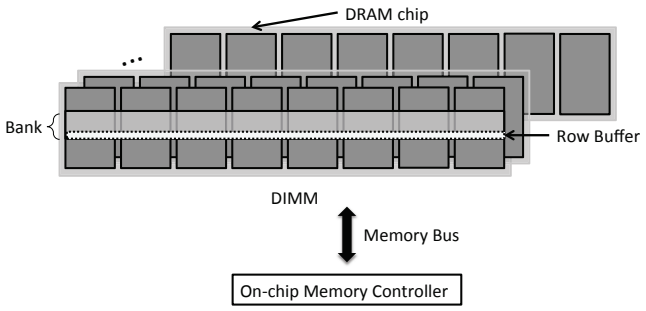


Figure 1: High-level organization of main memory (DRAM).

when optimizing for it. Third, an application that has already been optimized for cache locality may not necessarily exhibit good row buffer locality, and similarly, optimizing exclusively for row-buffer locality may not necessarily improve cache behavior. We are not aware of any prior compiler work that optimizes row-buffer locality of irregular applications. Motivated by this, we make the following contributions in this paper:

• We present a compiler-runtime cooperative data layout optimization approach that takes as input an irregular program that has already been optimized for cache locality and generates an output code with the same cache performance but better row-buffer locality (lower number of row-buffer misses).

• We present an alternate strategy which sacrifices some cache performance in order to further improve row-buffer performance (i.e., it trades cache performance for memory system performance). By carefully determining the amount of cache locality to sacrifice, this strategy improves the overall application performance.

• We perform a detailed, inspector/executor paradigm-based evaluation of these approaches using both an AMD Opteron based multicore system and a multicore simulator. The experimental results, collected using five real-world irregular applications, show that (i) conventional cache optimizations do not improve row-buffer locality significantly; (ii) our first approach achieves about 9.8% execution time improvement by keeping the number of cache misses the same as a cache-optimized code but reducing the number of row-buffer misses; (iii) our second approach achieves even higher execution time improvements (13.8% on average) by sacrificing cache performance for additional memory performance; We also provide experimental evaluation under different values of important architectural parameters such as the number of memory controllers, number of cores, row-buffer size and last level cache capacity, as well as different memory request scheduling algorithms.

2. BACKGROUND

2.1 On-Chip Caches

The cache is a small memory that stores data from frequently used main memory locations. Data is stored in chunks of several words at contiguous addresses called cache lines (or cache blocks). When new data is returned from a request and needs to be stored in the cache, some data already present in the cache may have to be evicted to make room. In a k-way *set-associative* cache, a memory block can be mapped to any of k different cache blocks. The cache replacement policy determines which of the k lines will be evicted and replaced. The state-of-the-art on-chip caches employ various policies such as Random Replacement [16], LRU (Least Recently Used) [16], SLRU [21], Round-robin [3], and PLRU [2]. Each of these policies has its own advantages and disadvantages. Further, except Random Replacement, these policies determine the cache block/line to be replaced by looking at the past references history. LRU is the most commonly used policy where the next cache block to be evicted is the one that has been least recently used. It requires several bits to track when each cache block is accessed. The number of these bits increases as the set-associativity increases. In this work, we focus on the on-chip caches that employ the LRU-like cache replacement policies.

Since accesses to words not currently present in the cache, called cache misses, take much longer to service than cache hits (where the data is present), varies approaches are proposed to try to minimize the number of cache misses by keeping useful data in the cache as much as possible. One way to achieve this is called *data reordering* or *layout reorganization*, which changes the positions of data elements stored in the memory. This technique can be used to put the data that will be used at around the same time on the same cache line; that way, most accesses to that data will hit in the cache.

2.2 Row-Buffers

The main memory consists of several DRAM chips logically organized into banks that are one row wide (which is usually equal to the page size). Each memory bank is accessed one row at a time, and the entire row must be copied into a small buffer called the row-buffer. In a DRAM, upon a read or write request, the data content of the requested row is read by sense ampliers and latched in the row-buffer. Many banks can be accessed in parallel since they each have their own row-buffer. If the memory uses an *open page policy*, the same row is kept in the row buffer after the initial request, allowing multiple accesses to the same row. The data in the row-buffer only changes if a different row is requested. Memory accesses that find their row already in the row buffer are called "row buffer hits" and can be significantly faster than "row buffer misses", which must wait for the row to be copied into the buffer. A row buffer miss can be even slower in the case of a conflict (when another row is already in the row buffer) because the previous row must be copied back before the new row can be brought in. One important factor affecting the row buffer hit rate is the hardware (memory request) scheduling policy [19] used by the memory controller to send requests into the channel. Row-buffer hit rates can also be affected by the data layout, in a similar way to caches. *Row-buffer locality* refers to the reuse of a memory row while its contents are in the row buffer. If two cache lines are frequently accessed together, their row-buffer locality will be better if they are placed in the same memory row (in order to improve row-buffer locality). Otherwise, they will experience row buffer conflicts. As compared to the vast literature on caches, there are only a few prior works [34, 14] on improving row-buffer locality from the compiler perspective, and none of them handles applications with irregular data accesses.

```
Real X(num_nodes), Y(num_edges);
Integer IA(num_edges, 2);
for (t = 1, t < T, t++) {
   for (i = 0, i < num_edges; i++) {
      X(IA(i, 1)) = X(IA(i, 1))  + Y(i);
      X(IA(i, 2)) = X(IA(i, 2))  - Y(i);
   }
}
```

Figure 2: Irregular code example.

2.3 Irregular Applications

In this work, our focus is on irregular application programs, in which a large fraction of data access has the form of $X(IA(f(\cdots)))$, where X and IA are two arrays called *host array* and *index array* [13], respectively, and $f(\cdots)$ is a function of enclosing loop indices and loop-independent constants. In these applications, the value of $IA(f(\cdots))$ is typically *not* known until run-time. Figure 2 illustrates the code structure of a sample irregular application. In this example, *IA(num_edges, 2)* lists the two nodes associated with each edge in a graph. The loop iterates over the edges of the graph and in each iteration, the values related to the two nodes of each edge will be updated accordingly. Traversing this index array will give us a sequence of data elements that will be accessed by the program. Due to the unknown value of index array IA at compile time, the accesses to the host array X are irregular, which makes the irregular code difficult to utilize caches efficiently.

There exist a number of work on improving the performance of irregular applications. One category is called *data reordering* [13, 15], which can be performed directly by rearranging elements in the host arrays. These work include, but not limited to, consecutive packing (CPACK) [13], Reverse Cuthill-McKee (RCM) [24], space filling curves (SFC) [25], recursive coordinate bisection (RCB) [9], multilevel graph partitioning (METIS) [18], and hierarchical clustering algorithm (GPART) [15]. The basic strategy adopted by these work is to relocate the elements such that the elements that tend to be accessed together in a short period of time become close in memory space. To implement this, the inspector-executor module [12, 13, 15] is often used. The inspector preprocesses the memory/data accesses, e.g., obtain the data access sequence of the program by traversing the index array; and the executor simply performs the optimization based on the information obtained by the inspector. There also exist several works on automatically generating inspectors and executors [15]. Figure 3 shows the structure of the inspector module used to implement the data reordering, where $Trans(X, Y)$ is the inspector inserted after each update of the index array. For different data reordering algorithms, the differences are mainly in $Trans(X, Y)$.

```
/* Executor*/
Real X(num_nodes), Y(num_edges);
Real X'(num_nodes), Y'(num_edges);
Integer IA(num_edges, 2);
for (t = 1, t < T, t++) {
   /*If it is time to update the
       interaction list */
   X', Y' = Trans(X, Y);
   for (i = 0, i < num_edges; i++) {
      X'(IA(i, 1)) = X'(IA(i, 1))  + Y'(i);
      X'(IA(i, 2)) = X'(IA(i, 2))  - Y'(i);
   }
}
```

```
/* Inspector*/
Trans(X, Y)
  for (i = 0, i < num_edges; i++) {
    /*data reordering
        algorithms*/
  }
return (X', Y')
```

Figure 3: Using the inspector module module for data reordering.

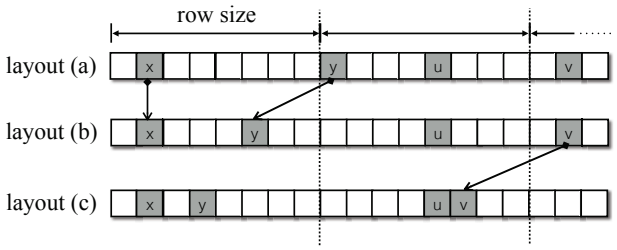


Figure 4: Motivational example. The grey boxes represent the data elements to which the accesses incur last level cache misses. Note that these figures show the layouts (not accesses), and each element shown here may be accessed multiple times during execution.

3. MOTIVATION

To our knowledge, all prior work that target irregular/sparse applications exclusively focus on improving cache locality, and they do not do anything special for row-buffer locality. To put cache locality and row-buffer locality into perspective, let us consider typical miss latencies of the corresponding components. A last level cache (L3) hit in our target AMD based architecture takes 28 clock cycles. A last level cache miss that hits in the row-buffer takes about 90 clock cycles, whereas a row-buffer miss costs more than 350 cycles. Therefore, eliminating one row-buffer miss is expected to bring much more benefits than eliminating a cache miss that would hit in the row buffer. That is to say, the overall performance of an application is not dictated only by the cache performance, but is also influenced by the row-buffer locality.

Figure 4 illustrates a motivational example. It contains three layouts for the same set of data elements. For simplicity, let us assume that, in the original layout (a), all the accesses to the highlighted data elements (grey boxes), x, y, u and v, incur last level cache misses.[1] In this layout, x and y are mapped to different rows. Therefore, if an access to x and an access to y incur successive cache misses (x is accessed first), then the access to y will also incur a row-buffer miss. In contrast, if we can generate a layout such that x and y are mapped to the same row[2], as shown in layout (b), then the access to y will incur a row-buffer hit. Further, if

[1]Unless specified, in the rest of our discussion, when we say "cache misses", we mean the "last level cache misses".

[2]When we say "placing two data elements into the same row", we mean "placing these two data elements into the memory blocks that will be mapped to the same row". Since the row size is usually equal to a page size, from the compiler perspective, this can be achieved by ensuring these two data elements have the same virtual page address. That is to say, we only care whether two data elements reside in the same page or not. As a result, we do not need to take the virtual-to-physical mapping into account.

this layout does not cause any additional cache misses, then the memory access latencies as well as the overall application execution time can be shorten. The layout generated in this way is called *conservative layout*. Compared to the original layout, the conservative layout has the same number of cache misses but fewer row-buffer misses.

Next, let us assume that there exist many accesses to u and v that incur successive cache misses (similar to the case of x and y in the above example), but layout (b) does not place u and v into the same row because doing so will cause additional cache misses. Can we take advantage of this data access pattern to generate a better layout than (b)? Recall that, normally the latency of a row-buffer miss is much higher than the latency of a cache miss. If we can generate a layout such that the overall increased cache miss latency is less than the overall reduced row-buffer miss latency, then the resulting memory access latency will be less than the original one. In other words, we sacrifice some cache performance in order to further improve row-buffer performance. For example, when we change the layout from (b) to (c), if only few accesses to v (which originally incur cache hits) turn into cache misses, and many accesses to v turn to row buffer hits (due to placing u and v into the same row), then with this new layout, the overall memory access latency can still be reduced. A layout generated in this way is called the *fine-grain layout* in this paper. Compared to the original layout, the fine-grain layout usually has more cache misses but fewer row-buffer misses.

The main contribution of this work includes two data layout reorganization (data reordering) approaches that derive conservative layout and fine-grain layout for irregular applications. The *input layout* of our framework could be the original layout or a layout obtained from prior data reuse optimizations that target minimizing cache misses (e.g., [15]). In other words, our starting/original layout can be one that has already been optimized for cache performance.

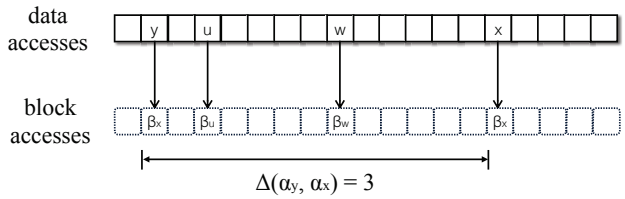
4. PRELIMINARIES

To simplify our explanation, we introduce the following notation:

- *Seq*: the sequence of data elements obtained by traversing the index array (based on the innermost loop(s)).
- α_x: the *access* to a particular data element x in *Seq*.
- $time(\alpha_x)$: the "logical time stamp" of α_x in *Seq*.
- β_x: the memory block where data element x resides.
- α_x^-: the "most recent access" to β_x before α_x.
- $Caches(\beta_x)$: the set of cache blocks to which β_x can be mapped in a k-way set-associative cache.

DEFINITION 4.1. ***Block Distance.*** *Given $Caches(\beta_x) = Caches(\beta_y)$, the block distance between α_x and α_y, denoted as $\Delta(\alpha_y, \alpha_x)$,[3] is the number of "distinct" memory blocks that are mapped to $Caches(\beta_x)$ and accessed during the time period between $time(\alpha_x)$ and $time(\alpha_y)$.*

[3]Note that, here we do not distinguish the appearance order of α_x and α_y in $\Delta(\alpha_y, \alpha_x)$, i.e., $\Delta(\alpha_y, \alpha_x) = \Delta(\alpha_x, \alpha_y)$.



(a) Block distance.

Figure 5: An example that illustrates the block distance concept.

As a special case, when $\alpha_y = \alpha_x^-$, $\Delta(\alpha_x^-, \alpha_x)$ is referred to as the *block reuse distance* of a_x. If $\beta_x = \beta_y$ and α_x is the first access (or α_y is the last access) to the memory block β_x in *Seq*, we define $\Delta(\alpha_y, \alpha_x)$ as ∞. Figure 5 gives an example of the block distance of α_x, where x and y reside in the same memory block ($\beta_x = \beta_y$), and the three memory blocks, β_x, β_u, and β_w, are mapped to the same set of caches ($Caches(\beta_x) = Caches(\beta_u) = Caches(\beta_w)$).

The block distance concept can be used to *determine whether a memory block can be reused or not in the cache*. Given a k-way set associative cache, if $\Delta(\alpha_y, \alpha_x) \leq k$ and $\beta_x = \beta_y$, then α_y ensures that β_x is in the cache at time $time(\alpha_x)$, and consequently, α_x will incur a cache "hit". Note that this conclusion holds true for most LRU-like cache replacement policies, as long as the victim cache block is selected among the k different cache blocks.

5. CONSERVATIVE LAYOUT

DEFINITION 5.1. ***Locality Set.*** *A set of data elements, denoted by Ω, forms a* locality set*, if and only if:*

1. $\forall\ x \in \Omega, \forall\ y \in \Omega,\ \beta_x = \beta_y$
2. $\forall\ x \in \Omega, \exists\ y \in \Omega, \exists\ \alpha_x, \exists\ \alpha_y$: $\Delta(\alpha_y, \alpha_x) \leq k$
3. $\forall\ x \notin \Omega, \forall\ y \in \Omega, \forall\ \alpha_x, \forall\ \alpha_y$: $\Delta(\alpha_y, \alpha_x) > k$

LEMMA 5.2. ***Non-increased Cache Misses****: Moving Ω from β_x to β_y will not increase the total number of cache misses in Seq if $Caches(\beta_y) = Caches(\beta_x)$.*

PROOF. Let α_i denote an arbitrary access in *Seq* that originally incurs a cache hit, i.e., $\Delta(\alpha_i^-, \alpha_i) \leq k$. Clearly, if $Caches(\beta_i) \neq Caches(\beta_x)$, then performing the above movement will not change $\Delta(\alpha_i^-, \alpha_i)$. If $Caches(\beta_i) = Caches(\beta_x)$, then after moving G from β_x to β_y, none of the distinct memory blocks in $Caches(\beta_x)$ that are accessed during time period $[time(\alpha_i^-), time(\alpha_i)]$ would change except that β_x will be replaced by β_y. Therefore, in this case, $\Delta(\alpha_i^-, \alpha_i)$ will not be increased, either. As a result, for any access in *Seq* that originally incurs a cache hit, the stated move operation will not change its block reuse distance, i.e., it will still incur a cache hit after this movement. □

To generate the conservative layout, the key is to ensure that the total number of cache misses is *not* increased. We found that, based on Lemma 5.2, this can be achieved by performing the data reordering at a locality set granularity. Specifically, our scheme consists of the following three steps, and the pseudo-code is given in Algorithm 1.

Algorithm 1 Conservative Layout Generation.

```
1: /*Identifying locality sets*/
2: for every access α_x in Seq do
3:   if list_{Caches(β_x)} is not full or (∀α_y ∈ list_{Caches(β_x)} :
     β_x ≠ β_y) then
4:     place α_x into list_{Caches(β_x)}
5:   else if ∃α_y ∈ list_{Caches(β_x)} : β_x = β_y then
6:     replace α_y with α_x in list_{Caches(β_x)}
7:     place x and y into the same locality set
8:   end if
9: end for
10: /*Constructing the interference graph*/
11: for every access α_x in Seq do
12:   if Δ(α_x^-, α_x) > k ∧ y == ∅ then
13:     y ← x
14:   else if Δ(α_x^-, α_x) > k ∧ y ≠ ∅ ∧ bank(x) = bank(y)
      then
15:     /* assume that x ∈ Ω_1 and y ∈ Ω_2 */
16:     set up edge e_{1,2} between Ω_1 and Ω_2
17:     weight(e_{1,2}) ++
18:   end if
19: end for
20: /*Assigning pages/rows and memory blocks*/
21: sort edges in non-increasing order based on their
    weights: e_1, e_2, ··· e_p
22: for every each pair of locality sets Ω_1 and Ω_2 on e_i do
23:   assign row(Ω_1) and row(Ω_2) to the same row.
24:   assign block(Ω_1) and block(Ω_2) to the memory blocks
      based on Lemma 5.2
25: end for
```

Step 1: Identifying the Locality Sets. We identify all the locality sets by traversing the index array (based on the innermost loop(s)). For each cache set, we maintain a list that stores the most recent accesses to k different memory blocks that are mapped to this cache set. In this way, the block distance between the current access α_x and any other access α_y on the list is never greater than k. Therefore, during this index array traversal, only when $\beta_x = \beta_y$ holds, we place x and y into the same locality set.

Step 2: Constructing the Interference Graph. Once the locality sets have been identified, our second step is to construct an *interference graph*, in which each node represents a locality set. If α_x and α_y are the two accesses that incur successive cache misses on Seq, and x and y are located in different pages/rows (in the original memory layout), then we set up an edge between the locality sets of x and y. The weight on this edge represents the total number of such α_x and α_y pairs. We construct this graph by making another pass over the index array. Each time when two accesses, α_x and α_y, incur successive cache misses, i.e., $\Delta(\alpha_x^-, \alpha_x) > k$ and $\Delta(\alpha_y^-, \alpha_y) > k$,[4] and x and y reside in different pages/rows, we increase the weight of the edge between the locality sets of x and y by 1. Overall, the weights on the edges of the interference graph evaluate how frequently the accesses to the data elements in two locality sets are missed successively.

Step 3: Assigning Pages/Rows and Memory Blocks. We first sort the edges in the interference graph in a non-increasing order of their weights. This is because, larger the weight of an edge that connects two locality sets, a higher fraction of row-buffer misses can be eliminated by placing these two locality sets into the same page/row. Next, we assign the same row to the locality sets connected by the edge with the largest weight. At this point, we also need to assign the memory blocks to the locality sets within that row. This is carried out by following the rule specified in Lemma 5.2. Once this assignment is finished, we update the available memory positions in this row, and then assign the row to the locality sets connected by the edge with the second largest weight, and so on. If the assignment fails due to the limited memory positions in a row, we simply skip this edge and proceed with the next one.

[4]We use the algorithm similar to Algorithm 1 to calculate $\Delta(\alpha_x^-, \alpha_x)$ and $\Delta(a_y^-, \alpha_y)$.

6. FINE-GRAIN LAYOUT

DEFINITION 6.1. ***Partition.*** *Given $x \in \Omega$, a partition for x is defined as a subset of Ω, denoted as P_x, where $x \in P_x$.*

To generate the fine-grain layout, our basic idea is that, whenever the accesses to two data elements (denoted as x and y) incur successive cache misses and x and y reside in different pages/rows, we try to find two *partitions* for x and y, denoted as P_x and P_y, respectively, such that, when placing P_x and P_y into the same page/row, the increased cache miss latency is less than the reduced row-buffer miss latency.

Our proposed data reordering scheme for generating the fine-grain layout consists of the following steps, and the corresponding pseudo-code is given in Algorithm 2.

Step 1: Constructing the Interference Graph. This step is slightly different from Step 2 described in Section 5. Now, each node in the interference graph represents a data element, instead of a locality set. If α_x and α_y are two accesses that incur successive cache misses on Seq, and x and y are located in different pages/rows, then we set up an edge between x and y. The weight on this edge represents the total number of such α_x and α_y pairs, which is also the number of row buffer misses that can be eliminated by placing x and y into the same row.

Step 2: Constructing the Locality Graph. Each locality set Ω corresponds to a locality graph, where each node represents a data element in Ω. We first identify all the locality sets using the strategy given in Algorithm 1, and then construct the locality graph for each locality set as follows. For any access α_u whose block reuse distance is exactly k, if there exist α_x and α_y within time slot $[time(\alpha_u^-), time(\alpha_u)]$, such that x, y and u belong to the same locality set, then we increase the weight of the edge between x and y by 1. As a result, *if we move all the elements in a partition for x, denoted as P_x, to another memory block β'_x, such that $Caches(\beta'_x) = Caches(\beta_x)$, then the number of increased cache misses is at most equal to the sum of the weights of the edges connected to x in the locality graph.* Since each newly-increased cache miss can potentially lead to a row-buffer miss as well, this weight also represents the maximum number of increase in the row-buffer misses. Constructing all these locality graphs requires one additional pass over the index array.

Step 4: Finding Partitions. We first sort the edges in the interference graph in a non-increasing order of their weights, and then process the data elements on each edge in this order. Specifically, given x and y on an edge with

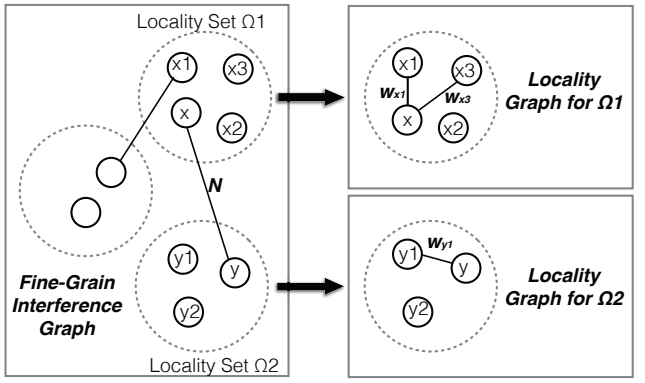


Figure 6: Example of finding the partitions that satisfy Condition (1). In each locality set, the black nodes are considered to be in the same partition (for each locality graph).

weight N, which represents the number of row-buffer misses that can be eliminated by placing x and y into the same row, we first consider isolating x and y from their locality sets, i.e., placing only x into P_x, and only y into P_y. Let N_{ch} and N_{rb} denote the total number of cache misses and row-buffer misses that could be increased by placing P_x and P_y into the memory blocks β'_x and β'_y, respectively, where $Caches(\beta'_x) = Caches(\beta_x)$ and $Caches(\beta'_y) = Caches(\beta_y)$, and δ_{ch} and δ_{rb} denote the miss latencies in the cache and row-buffer, respectively; then if these two partitions causes less overall access latency than the original, we should have:

$$(N - N_{rb}) * \delta_{rb} > N_{ch} * \delta_{ch}. \quad (1)$$

If Condition (1) is not satisfied, we then consider placing the data elements connected to x (in the locality graph), together with x, into P_x, and the data elements connected to y, together with y, into P_y. We repeat this process until Condition (1) is satisfied.[5] In the worst case when no such partitions can be found, we have $N_{ch} = N_{rb} = 0$, which makes Condition (1) still hold. At this point, the remaining question is how to determine N_{ch} and N_{rb}. Assuming that, in the locality graph, x is connected to $x_1, x_2, \cdots, x_m$, with the weights $w_{x_1}, w_{x_2}, \cdots, w_{x_m}$; and y is connected to $y_1, y_2, \cdots, y_n$, with the weights $w_{y_1}, w_{y_2}, \cdots, w_{y_n}$; then we have $N_{ch} = \sum_{i=1}^{m} w_{x_i} + \sum_{i=1}^{n} w_{y_i}$. Since N_{ch} is the upper limit for N_{rb}, in our current implementation, we conservatively have $N_{rb} = N_{ch}$.

Figure 6 illustrates an example for finding the partitions that satisfy Condition (1). In this example, we assume that, in the fine-grain interference graph, the edge between x and y has the highest weight N, and denote the locality sets of x and y as Ω_1 and Ω_2. In order to find the partitions for x and y, we first let $P_x \leftarrow x$ and $P_y \leftarrow y$, i.e., the initial partitions for x and y only contain themselves. At this point, we have $N_{ch} = w_{x_1} + w_{x_3} + w_{y_1}$. Assuming that Condition (1) cannot be satisfied based on this value, then we place all the neighbors of x and y into their partitions, i.e., $P_x = \{x, x_1, x_3\}$ and $P_y = \{y, y_1\}$. Now we have $N_{ch} = 0$, and therefore Condition (1) is satisfied, and the partitions of x and y are found.

Step 4: Assigning Rows and Memory Blocks. Once this process completes, we assign the rows and memory blocks to the partitions (or locality sets in the worst case) of x and y as described in Lemma 5.2.

Algorithm 2 Fine-Grain Layout Generation.

```
1: /*Constructing the interference graph*/
2: for every access α_u in Seq do
3:   if Δ(α_u^-, α_u) = k, u ∈ Ω then
4:     for each access in [time(α_u^-), time(α_u)] do
5:       α_x= current_access
6:       if Caches(β_x) = Caches(β_u) then
7:         /* tmp(β_x) records the previous access */
8:         if tmp(β_x) = ∅ then
9:           tmp(β_x) = x
10:        else
11:          y = tmp(β_x)
12:          set up edge e_{x,y} between x and y
13:        end if
14:      end if
15:    end for
16:  end if
17: end for
18: /* Assigning rows and memory blocks*/
19: sort edges in non-increasing order based on their
    weights: e_1, e_2, ··· e_p
20: for every each pair of x and y on e_i do
21:   P_x ← {x}
22:   P_y ← {y}
23:   N ← w(e_{x,y})
24:   while (N − N_rb) * δ_rb > N_ch * δ_ch do
25:     P_x ← P_x ∪ all neighbors of x in the locality graph
26:     P_y ← P_y ∪ all neighbors of y in the locality graph
27:   end while
28:   assign row(P_x) and row(P_y) to the same row.
29:   assign block(P_x) and block(P_y) to the memory blocks
      based on Lemma 5.2
30: end for
```

7. DISCUSSION

Sequential code vs. parallel code: So far our discussion is based on the assumption that the sequence of data

[5]We are not trying to obtain the optimal solution w.r.t Condition (1). Instead, our point is that, as long as this condition is satisfied, the generated layout will be "better" than the original one.

accessed by the program, Seq, is equal to the sequence of data visited by traversing the index array. Although in a single-threaded application, these two orders are expected to be the same, in a multi-threaded application, the unpredictable interferences across threads make these two orders usually different. However, we want to point out that, assuming these orders equal may be a reasonable approximation. This is because, the foundation of our proposed schemes is the relationship between the block reuse distance of each access and the value of k. That is to say, we only need to know whether this block distance is greater than k or not, and do not need to know its exact value. Therefore, they have certain tolerance to the mismatch between the "assumed" and "actual" data access orders. For example, when determining $\Delta(\alpha_x, \alpha_y)$ (assuming that $\beta_x = \beta_y$), we do not need to figure out the exact position of a_y in the actual data access order, instead, we only need to know if a_y appears in the most-recent k accesses before a_x or vice versa. Therefore, the "assumed" data access order can be considered as a reasonable approximation of "the actual" data access order that will be observed at run time. There also several existing works on how to predict the data access sequence of a multithreaded application more accurately [10, 33], which could be our future research direction.

Complexity analysis: Next we compare the algorithmic complexities of our data ordering scheme with existing schemes that target optimizing cache behavior of irregular applications. We assume that the number of data elements in the data access sequence is N, the length of the index array is E, and the cache size is C. For typical graphs, we have $N >> C$ and $N > E$. The consecutive packing (CPACK) has a cost of $O(E)$, since it rearranges the data elements based on their "first-touch" order and requires one pass through the data access sequence. $GPART$ has a cost of $O(E * g(C))$, where $g(C)$ is a constant function greater than 1, since the number of clustering passes is dependent on the cache size, *not* the input data size. As a result, GPART is only a small constant factor more expensive than CPACK. RCB has the complexity of $O(N(log_2 N)^2)$, and METIS has the cost $O(2Nlog_2 N) + O(E)$ [15]. In comparison, assuming that the inference graph has E' edges, as can be observed in Algorithms 1, conservative layout has the complexity of $O(N) + O(E_1)$ since each of the first two steps requires one pass over the index array ($O(N)$), and the last step requires one traversal of the inference graph ($O(E')$). For the fine-grain layout, let us assume that the inference graph has E_1 edges and the locality graph has E_2 edges and V nodes. Similar to the conservative layout, each of the first two steps requires one pass over the index array ($O(N)$), and the last step has a cost of $O(E_1(V + E_2))$. Therefore, the total cost of generating fine-grain layout is $O(N) + O(E_1(V + E_2))$, which is higher than the cost of generating the conservative layout, and both of our two data reordering schemes have lower complexities than RCB and METIS, but higher complexities than CPACK and GPART.

Implementation overheads: The inspectors are expensive, but their "one-time" cost can be amortized over a long execution time. For example, the inspectors can be inserted outside the loop, and then, execute once before the computation begins, benefiting computations throughout the whole loop. When access patterns change, the inspectors may be invoked, but they need not to be rerun each time the access patterns change. Without rerunning the inspectors, the cache performance may degrade, but the application code still generates correct results. Similar to the prior work, we only modify the implementation of the inspector to embed our data reordering strategies.

Data reordering itself also introduces overheads during the execution (runtime), which mainly comes from two sources. The first is the overhead of the data reordering itself, which can be reduced by adjusting the number of edges in the inference graph and locality graph that need to be traversed. The compiler needs to ensure that this cost does not outweigh any performance gain by data reordering. The second overhead comes from the data redirection after data reordering, which maps the old memory location (before data reordering) to the new memory location (after data reordering). Since each access to a transformed array needs to be redirected to the access to the new memory location, this overhead can be quite expensive. Discussion about how to reduce this type of overhead can be found in [13].

Table 1: Benchmark codes used in our evaluation.

Name	Input Size	L3 Miss Rate	Row-Buffer Miss Rate	Execution Time
PSTT	427.6MB	18.1%	29.6%	9.6sec
PaSTiX	511.6MB	24.3%	41.7%	8.8sec
SSIF	129.3MB	13.7%	24.4%	4.3sec
PPS	738.2MB	21.4%	33.1%	11.1sec
REACT	1.2GB	28.6%	46.9%	13.7sec

Table 2: Architecture specification.

CPU	48 cores; 4 sockets (12 cores/socket); 4 memory controllers/socket (with FR-FCFS scheduling policy); 2.6GHz AMD Opteron cores;
Cache Hierarchy	64KB 2-way associative per core L1 (3 cycles); 512KB 16-way associative per core L2 (12 cycles); 12MB per socket 6-way associative shared L3 (28 cycles)
Memory System	DDR3 1,866MHz DRAM; 8 banks per channel; 8KB row-buffers;

8. EXPERIMENTAL EVALUATION

8.1 Implementation, Benchmarks and Setup

We implement our proposed schemes using the Open64 compiler [4]. In our implementation, the compiler generates run-time library calls to the *inspector module* to process memory access patterns at run-time and identify non-local data needed by each processor, and determines when the inspectors must be rerun if the memory access patterns change in the program code. Specifically, the compiler first makes a pass through the program to identify the index array. Then it finds appropriate locations where inspectors can be inserted. To do this, a top-down traversal of the AST is performed to identify the topmost blocks that contain the writes to the index arrays, but do not contain corresponding irregular array computations. The inspectors will be inserted in these blocks. When the inspector is executed at run-time, it rearranges the contents of the host arrays to generate the desired data reordering by using Algorithms 1 and 2 presented earlier. Later, the executor (the original loop body) will access these reordered data, which helps us to exploit the cache and row-buffer locality.

For each application code in our experimental suite, we tested four different versions: *Original* (the original code without any modification, compiled using the default -O2 compiler flag), *Cache-Optimized* (a version which is optimized using a previously-proposed cache locality enhancement technique (GPART [15]), followed by all data locality optimizations turned on using the -O3 compiler flag – these optimizations include loop permutation as well as tiling among others), *Conservative* (our conservative layout optimization strategy explained in Section 5), and *Fine-Grain* (our fine-grain layout optimization strategy explained in Section 6). We want to point out that, other algorithms such METIS [18] and RCB [9], as mention in Section 7, have significantly higher runtime overheads than GPART and CPACK, and they are quite expensive when used for cache optimizations [15]. Therefore, in a scenario like ours where the layout needs to be modified during the course execution, there is a motivation for reordering-based techniques. In addition, GPART performs better than METIS for most cases [15]. And, since our preliminary experiments also indicated the same, we chose GPART for comparison. However, METIS and similar partitioning tools have a very important role in sparse/irregular application codes. For example, in a cluster-environment, the target-graph can be partitioned using METIS to minimize inter-node communication, and then the local-portion of each node can be laid out in memory using our approach. In that case, METIS and our approach each addresses different aspects of locality: one across nodes (minimizing communication in distributed-memory-space), and the other across cores in a node (cache locality as well as row-buffer locality). In addition, we did not turn on hardware-prefetching in our experiments presented below as it does not bring much benefit in irregular-applications. The experiments with our approach with hardware-prefetching turned-on did not change our savings much (it reduced the additional benefits brought by our approach by less than 1%). Note also that, if the prefetching-algorithm is exposed to our approach, we can adapt our approach by fine-tuning the block-distance-calculation.

For our experimental evaluations, we used five applications: PSTT (parallel sparse FFT) [5], PaSTiX (a high performance parallel solver for very large sparse linear systems based on direct methods) [6], SSIF (sparse symmetric indefinite factorization) [26], PPS (a persistent, pervasive surveillance code), and REACT (Jacobian based combustion modeling code). The last two applications are written by us. We want to emphasize that the first three of these applications are quite large programs (each having more than 5,000 lines excluding library calls). Further, all five applications are memory-intensive and operate on large data structures, creating a high volume of off-chip data traffic. Table 1 gives the salient features of these applications; the last three columns show, respectively, the last level cache miss rate, the row-buffer miss rate, and parallel execution time on our 12 cores (1 socket of) AMD based system (explained below). It is important to note that, as far as row-buffer misses are concerned, these applications exhibit a variety (ranging between 24.4% and 46.9%). Overall, however, these row-buffer miss rates are quite high, and indicate poor row-buffer locality. There are three main reasons for this. First, irregular application inherently have poor locality within memory rows. Second, techniques such as cache block interleaving (i.e., distributing physical memory across controllers at a cache block granularity), while good for memory level parallelism, hurt row-buffer locality as consecutive cache blocks go to different banks, reducing the amount of row buffer reuse. Third, running an irregular application in a multithreaded fashion further spreads data access across the memory space, and this in turn also has a negative impact on row-buffer locality.

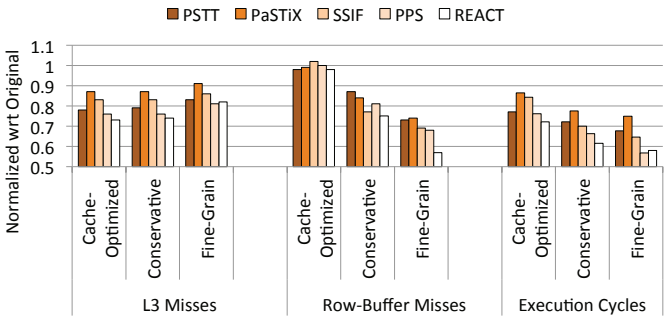


Figure 7: Results with AMD based multicore system.

In this work, we performed two types of evaluations. First, we measured the impact of our optimizations on an AMD Opteron based multicore with 8 cores, 2.6GHz core speed and a total of 12MB L3 last-level cache. The relevant important characteristics of this system are given in Table 2. All the cache and row-buffer statistics reported in this table as well as those presented in the remainder of this section for this AMD based machine are collected by monitoring the Opteron CPU and memory controller performance counters. Second, we also simulated our approach on GEM5 multicore simulator [1]. The reason for the simulation based experiments is this: while the AMD multicore experiments clearly show the impact of our proposed row-buffer optimization, it is not possible in a real hardware to change different parameters and test how our approach would perform, for instance, when we increase the number of memory controllers, change the row-buffer size, bank count, etc. To carry out all these sensitivity studies, we also conducted a simulation-based evaluation.

8.2 Results with AMD Opteron

Figure 7 gives the results collected on one socket (12 cores) of our AMD Opteron based system (multi-socket results will be presented later). The y-axis in this plot represents normalized value (L3 miss rates, row-buffer miss rates, and execution cycles) with respect to Original (the absolute values of these metrics with Original are listed in Table 1). It can be seen that all three optimized versions (Cache-Optimized, Conservative and Fine-Grain) improve – except in two cases –over Original in all metrics quantified. More specifically, Cache-Optimized improves cache performance (with a geometric average of 20.6%) but does not bring much benefit as far as row-buffer locality is concerned, indicating that *conventional cache optimizations may not be very effective when it comes to row-buffers*. This version brings an average of 20.8% reduction in execution cycles compared to Original. Conservative on the other hand exhibits a different behavior. While maintaining the same cache performance as Cache-Optimized (difference between their cache performance is around 1% in some applications), it cuts the row-

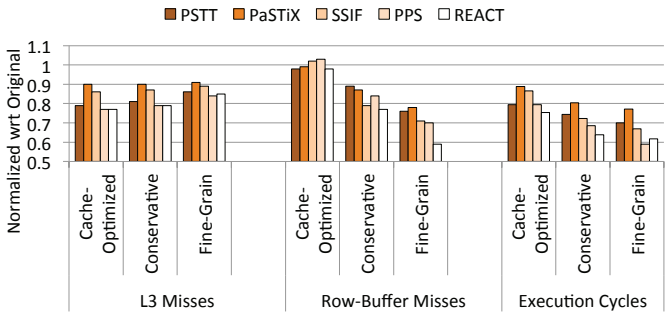


Figure 8: Simulation results.

buffer misses significantly (an average of 19.2% reduction over Original). Finally, Fine-Grain sacrifices some cache performance but in return achieves an average row-buffer miss reduction of 31.8%. At the end, Conservative and Fine-Grain improve 30.6% and 35.6%, respectively, over Original, in terms of execution time. These results demonstrate that *Fine-Grain is very successful in trading cache performance for better overall memory performance*, and selecting the right tradeoff point (which our approach tries to do) can result in significant improvements in execution time as well.

8.3 Simulation Results

We also evaluate the sensitivity of proposed scheme to different important architectural parameters and used a simulator (Gem5 [1]) for that. Specifically, there are four main parameters we wanted to study in detail: row-buffer size, number of memory controllers, number of cores, and L3 capacity. To carry out this study, we first modeled our AMD machine in the simulator as accurately as we can (using the values in the AMD system for all major parameters, the same compiler and the same thread mapping) and tried to replicate the results collected in the real system. The results presented in Figure 8 are a bit different from those in Figure 7 – as far as absolute values are concerned – but the general trends are similar.

In the remainder of this subsection, we present results for only one of our applications (PPS) since the general observations/trends with the others were similar. Figure 9(a) presents the sensitivity of our approach to row-buffer size. Remember that our default buffer size was 8KB (see Table 2). We see that our approaches (Conservative and Fine-Grain) perform better with larger row-buffer sizes. This is because a larger row-buffer size give more flexibility to our approach in reorganizing data elements. While Cache-Optimized also takes advantage of a larger row-buffer size, its improvement is far below than those of Conservative and Fine-Grain, and quickly saturates. Next, we report in Figure 9(b) the impact of varying the number of memory controllers, per socket. Again (and as would be expected), Conservative and Fine-Grain take much better advantage of the increased number of memory controllers. This is mainly because a larger number of controllers means that accesses are spread more in the memory space and this presents more scope to our approaches for optimization. We observe a similar trend when the number of cores per socket is increased (Figure 9(c)) for a similar reason – wider distribution of data accesses in the memory space with increased number

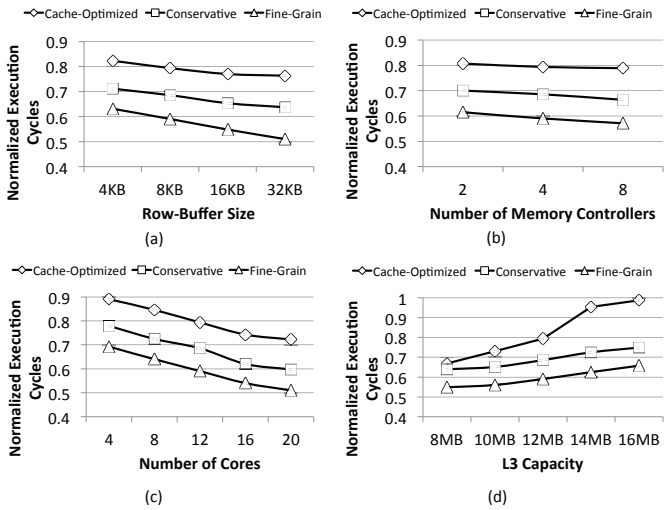


Figure 9: Results of sensitivity analysis (normalized w.r.t original).

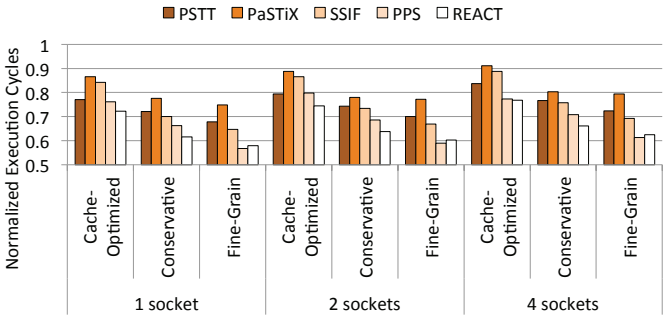


Figure 10: Multi-socket evaluation on AMD (normalized w.r.t original).

of cores. Our last sensitivity study involves changing L3 capacity (from its default value of 12MB). The results plotted in Figure 9(d) show that as cache capacity per core gets reduced, we can expect better savings from our layout optimization approaches.

An important conclusion from these sensitivity experiments is that our approach performs better with larger number of cores, larger number of memory controllers, increased row-buffer sizes and reduced last level cache space per core (not the cumulative LLC size), all of which are current trends in computer architecture. In other words, we can expect our optimizations to continue to be effective in the long run.

8.4 Multi-Socket Evaluation in Opteron

Recall that in our evaluation on the AMD based machine presented above, we used a single socket. Figure 10 plots normalized execution time (with respect to Original) with 2 sockets and 4 sockets as well. In the 2 socket case, each application is parallelized over 24 cores and in the case of 48 cores. The results indicate that, while savings achieved by our approaches, Conservative and Fine-Grain, get a bit reduced with increased socket count, overall they still achieve signifi-

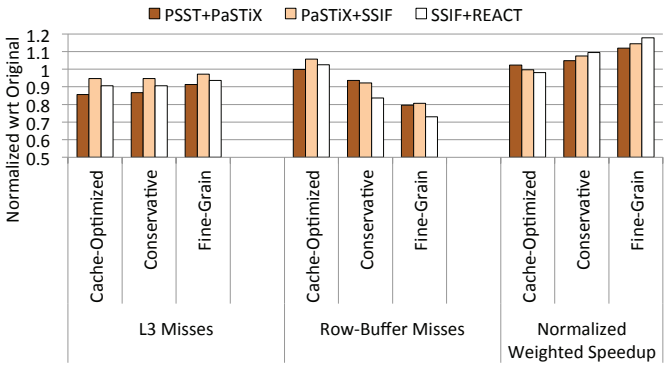


Figure 11: Multiprogrammed workload evaluation on AMD.

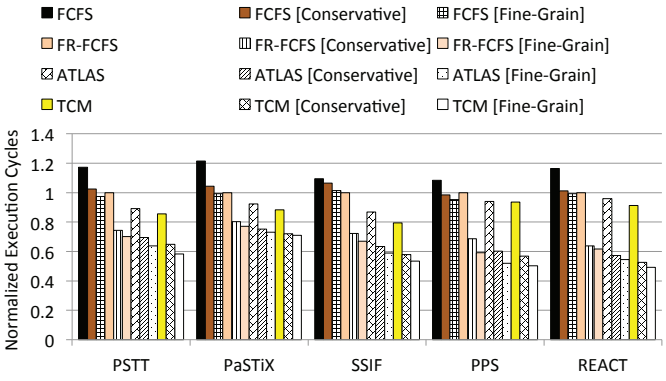


Figure 12: Results with different memory scheduling algorithms (normalized w.r.t FR-FCFS (with original)).

cant savings (the geometric mean of execution improvement is 28.3% and 33.3% in the 2-socket case, and 26.1% and 31.2% in the 4-socket case). The slight reduction when using 2/4 sockets is probably due to non-local accesses (those going from one socket to another) playing a major role in shaping the overall memory performance, and reducing the effectiveness of improved row-buffer behavior.

8.5 Multiprogrammed Workload Evaluation in Opteron

Our last set of experiments focus on the execution of a multiprogrammed workload of multithreaded applications. In this experiment, we formed three workloads, each consisting of two applications executing on 6 cores of a socket. That is, two applications share the cores in a socket equally. It should be noted that we can expect the effectiveness of our approaches to reduce in this case, primarily because both applications in a workload will share the same 4 memory controllers (hence same set of row-buffers) in the socket, and this will cause extra row-buffer conflicts/misses for each application (due to interference). The metric we use to represent performance is "combined speedup", which is basically the average speedup the two applications achieve over their Original versions. The results plotted in Figure 11 show that, when averaged over all three workloads, Conservative and Fine-Grain achieve 7.3% and 14.7% performance improvement. Therefore, it can be concluded that our approach is very effective even if multiple applications share the same set of memory controllers.

8.6 Results with Different Memory Schedulings

DRAM is a major resource shared among cores in a multicore machine, and memory requests from different threads can interfere with each other when accessing this shared resource. Motivated by this, there is a plethora of hardware-based memory request scheduling algorithms oriented towards reducing this potential interference and increase the overall throughput of the memory system. As stated in Table 2, the default memory scheduling we used so far (and the dominant scheduling algorithm in commercial chips) is FR-FCFS, which works by implementing first-come, first-served (FCFS) strategy in serving the memory requests, except when there are memory requests in the queue that target the current memory row in the row-buffer. That is, FR-FCFS give priority to requests that can reuse the row-buffer contents; when no such request can be found in the queue, it resorts to FCFS. Recent years have witnessed more sophisticated memory scheduling policies that are very aggressive in prioritizing memory requests and coordinating memory controllers to further improve memory throughput. Two such schemes are ATLAS [19] and TCM [20]. While ATLAS periodically orders the threads based on the service they have attained from the memory controllers so far, and prioritize those threads that have attained the least service over others in each period, TCM divides the threads into two separate clusters and employ different memory request scheduling policies in each cluster. Using GEM5, we evaluated our conservative and fine-grain layout optimization strategies under pure FCFS (without any prioritization), ATLAS and TCM, and plot results in Figure 12. Note that the bar marked as "FR-FCFS" indicates the execution of Original benchmark under the FR-FCFS policy, and represents the base case that we have used so far. Several interesting observations can be made from this plot. First, the performance of FCFS powered with our layout optimization algorithms are competitive with FR-FCFS with the original codes. Given the fact that going from FCFS to FR-FCFS in hardware requires a lot of additional cost and complexity (as one need to check all the entries in the memory queue to see whether they access the row currently in the row-buffer), we believe that our software based strategy provides a cost effective alternative to hardware optimization. Second, we see that using our approach with FR-FCFS generates better results than using the original codes with ATLAS or TCM. Third, our layout reorganization strategy improves the performance of ATLAS and TCM significantly, and in fact, the highest performance improvement is achieved by TCM (Fine-Grain). In principle, these savings can be further increased by making our approach aware of the underlying memory scheduling algorithm; we postpone the investigation of this problem to a future study.

9. RELATED WORK

There exist many work related to row buffer locality [19, 20, 32, 39, 29, 17, 7, 43]. Lee et. al. [23] tries to reduce interference between reads and writes in the memory channel. They take advantage of row buffer hits by scheduling write backs that will hit in the row buffer before they are evicted

from the cache. Awasthi et al. [8] proposed an Access Based Predictor, which keeps a history of the number of accesses to each DRAM page in order to predict how long the row buffer should remain open when that page is present. Yoon et al. [38] avoid the high cost of phase-change memory (PCM) row buffer misses in hybrid memories by moving memory blocks that frequently miss in the row buffer into DRAM. Their scheme also considers row buffer locality as a factor in deciding which blocks to move. Meza et al. [27] study the benefits of smaller row buffer sizes for non-volatile memory. Zhang et al. [41] described causes of row buffer conflicts when using a page interleaving policy. Their work improved row buffer hit rates using permutations that preserve data locality while distributing pages to reduce conflicts. Similarly, [36] tried to improve row buffer hit rates by decreasing the OS page size and moving frequently accessed pages into the same row. Most of these prior works concerning the row buffer locality target hardware, while our work uses compiler optimizations. There have not been as many compiler schemes for row buffer locality. One work [34] uses loop transformations to improve spatial locality of pages and therefore row buffer reuse. Unlike our approach, their optimization relies on dependences between data elements. Also, they target single-thread locality rather than taking into account the row buffer locality of the entire application. Another compiler scheme [14] targeted the row buffer locality of regular applications with affine loops. In contrast, we target irregular applications and try to improve row-buffer locality by trading cache performance for the overall memory performance.

Optimizations for irregular applications often target data layout. Data clustering is an NP-hard problem, as shown in early work by Thabit [37]. Furthermore, Petrank and Rawitz [30] demonstrated that data placement to minimize cache misses is an inapproximable optimization problem. Zhong et al. [42] introduce a metric of the closeness of data accesses based on LRU stack distance called reference affinity. Zhang et al. [40] looked at reference affinity from both a theoretical and a practical perspective. Our concept of a locality set differs from the affinity group described by these papers in that locality set describes a set of elements in the cache with good locality, while an affinity group is built based on the reference affinity metric. More practical approaches to the problem of irregular accesses include [13, 15, 25, 28, 35]. However, all these studies either try to improve parallelism, or reduce runtime overheads, or improve cache performance. None of them addresses row-buffer locality. [15] proposed a hierarchical graph clustering algorithm (GPART) to improve cache locality.

10. CONCLUSIONS

In a system that employs open page policy, exploiting row-buffer locality can be critical. The novel contribution of this paper is a compiler-directed data layout reorganization scheme with the goal of improving row-buffer locality of large, irregular applications. In particular, we propose, implement and test two alternate layout reorganization schemes, the first trying to improve row-buffer hits without negatively affecting the cache locality (hit/miss statistics) of a cache-optimized code, whereas the second one trading off cache hits to further improve the row-buffer hits, and eventually the overall application performance.

Acknowledgment

This research is supported in part by NSF grants #0963839, #1302557, #1213052, #1017882, and #1205618, a grant from INTEL an a grant from MICROSOFT.

11. REFERENCES

[1] "Gem5," *http://gem5.org*.

[2] "Intel pentium 4 and intel xeon processor optimization, reference manual," *http://developer.intel.com*.

[3] "Intel Xscale core, developer's manual," *http://developer.intel.com*.

[4] "Open64," *http://www.open64.net*.

[5] "Spiral: Software/hardware generation for dsp algorithms," *http://www.spiral.net/*.

[6] "The PaSTiX Library," *http://pastix.gforge.inria.fr/*.

[7] R. Ausavarungnirun, K. Kai-wei, C. Lavanya, S. Gabriel, H. Loh, and O. Mutlu, "Staged memory scheduling: achieving high performance and scalability in heterogeneous systems," *In Proceedings of the International Symposium on Computer Architecture*, 2012.

[8] M. Awasthi, D. W. Nellans, R. Balasubramonian, and A. Davis, "Prediction based DRAM row-buffer management in the many-core era," *In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[9] M. Berger and S. Bokhari, "A partitioning strategy for non-uniform problems on multiprocessors," *IEEE Trans. Computers*, 1987.

[10] P. Boonserm, B. Wang, S. See, and T. Achalakul, "Improving data processing time with access sequence prediction," in *Proceedings of the International Conference on Parallel and Distributed Systems*, 2012.

[11] P. Carribault, S. Zuckerman, A. Cohen, and W. Jalby, "Deep jam: Conversion of coarse-grain parallelism to instruction-level and vector parallelism for irregular applications," *In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[12] R. Das, M. Uysal, J. Saltz, and Y. shin Hwang, "Communication optimizations for irregular scientific computations on distributed memory architectures," *Journal of Parallel and Distributed Computing*, 1993.

[13] C. Ding and K. Kennedy, "Improving cache performance in dynamic applications through data and computation reorganization at run time," *In Proceedings of the Conference on Programming Language Design and Implementation*, 1999.

[14] W. Ding, J. Liu, K. Mahmut, and M. J. Irwin, "Reshaping cache misses to improve row-buffer locality in multicore systems," *In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2013.

[15] H. Han and C.-W. Tseng, "Exploiting locality for irregular scientific codes," *IEEE Trans. Parallel Distrib. Syst.*, 2006.

[16] J. L. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2003.

[17] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," *In Proceedings of the International Symposium on Computer Architecture*, 2008.

[18] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, 1998.

[19] Y. Kim, D. Han, O. Mutlu, and M. Harchol-balter, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," *In Proceedings of the International Symposium On High Performance Computer Architecture*, 2010.

[20] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," *In Proceedings of the International Symposium on Microarchitecture*, 2010.

[21] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," *Computer Architecture*, 1981.

[22] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval, "How much parallelism is there in irregular applications?" *In Proceedings of the ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 3–14, 2009.

[23] C. J. Lee, V. Narasiman, E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Dram-aware last-level cache writeback: Reducing write-caused interference in memory systems," *HPS Technical Report*, 2010.

[24] W. Liu and A. Sherman, "Comparative analysis of the cuthill-mckee and the reverse cuthill-mckee ordering algorithms for sparse matrices," *SIAM Journal on Numerical Analysis*, 1976.

[25] J. Mellor-Crummey, D. Whalley, and K. Kennedy, "Improving memory hierarchy performance for irregular applications," *In Proceedings of the International Conference on Supercomputing*, 1999.

[26] O. Meshar, D. Irony, and S. Toledo, "An out-of-core sparse symmetric-indefinite factorization method," *ACM Trans. Math. Softw.*, 2006.

[27] J. Meza, J. Li, and O. Mutlu, "Evaluating row buffer locality in future non-volatile main memories," *SAFARI Technical Report*, 2012.

[28] N. Mitchell, L. Carter, and J. Ferrante, "Localizing non-affine array references," *In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 1999.

[29] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," *In Proceedings of the International Symposium on Microarchitecture*, 2007.

[30] E. Petrank and D. Rawitz, "The hardness of cache conscious data placement," *In Proceedings of the Conference on Principles of Programming Languages*, 2002.

[31] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," *In Proceedings of the Conference on Programming Language Design and Implementation.*, 2011.

[32] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," *In Proceedings of the International Symposium on Computer Architecture*, 2000.

[33] M. F. Sakr, S. P. Levitan, D. M. Chiarulli, B. G. Horne, and C. L. Giles, "Predicting multiprocessor memory access patterns with learning models," *In Proceedings of the International Conference on Machine Learning*, 1997.

[34] J. Shin, J. Chame, and M. W. Hall, "A compiler algorithm for exploiting pagemode memory access in embeddeddram devices," *In Proceedings of the Workshop on Media Streaming Process*, 2002.

[35] M. M. Strout, L. Carter, and J. Ferrante, "Compile-time composition of run-time data and iteration reorderings," *In Proceedings of the Conference on Programming Language Design and Implementation*, 2003.

[36] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, "Micro-pages: increasing dram efficiency with locality-aware data placement," *In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[37] K. Thabit, "Cache management by the compiler," Ph.D. dissertation, Rice University, Houston, TX, USA, 1982.

[38] H. Yoon, J. Meza, R. Ausavarungnirun, R. Harding, and O. Mutlu, "Row buffer locality-aware data placement in hybrid memories," *SAFARI Technical Report*, 2011.

[39] G. L. Yuan, A. Bakhoda, and T. M. Aamodt, "Complexity effective memory access scheduling for many-core accelerator architectures," *In Proceedings of the International Symposium on Microarchitecture*, 2009.

[40] C. Zhang, C. Ding, M. Ogihara, Y. Zhong, and Y. Wu, "A hierarchical model of data locality," *In Proceedings of the Conference on Principles of Programming Languages*, 2006.

[41] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," *In Proceedings of the International Symposium on Microarchitecture*, 2000.

[42] Y. Zhong, M. Orlovich, X. Shen, and C. Ding, "Array regrouping and structure splitting using whole-program reference affinity," *In Proceedings of the Conference on Programming Language Design and Implementation*, 2004.

[43] Z. Zhu and Z. Zhang, "A performance comparison of dram memory system optimizations for SMT processors," *In Proceedings of the International Symposium on High-Performance Computer Architecture*, 2005.