A FRESH LOOK AT DATA LOCALITY ON EMERGING MULTICORES AND MANYCORES

A Dissertation in
Computer Science and Engineering
by
Wei Ding

© 2014 Wei Ding

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2014
The dissertation of Wei Ding was reviewed and approved* by the following:

Mahmut Taylan Kandemir  
Professor, Department of Computer Science and Engineering  
Dissertation Advisor, Chair of Committee  

Mary Jane Irwin  
Evan Pugh Professor and A. Robert Noll Chair, Department of Computer Science and Engineering  

Padma Raghavan  
Professor, Department of Computer Science and Engineering  

Dinghao Wu  
Assistant Professor, College of Information Sciences and Technology  

Lee Coraor  
Associate Professor, Director of Academic Affairs, Department of Computer Science and Engineering  
Graduate Program Chair  

*Signatures are on file in the Graduate School.
Abstract

The emergence of multicore platforms offers several opportunities for boosting application performance. These opportunities, which include parallelism and data locality benefits, require strong support from compilers as well as operating systems. However, architectural abstractions relevant to memory system are scarce in current programming and compiler systems. In fact, most compilers do not take any memory system specific parameter into account even when they are performing data locality optimizations. Instead, their locality optimizations are driven by rule-of-thumbs such as “maximizing stride-1 accesses in innermost loop positions”. There are a few compilers that take cache and memory specific parameters into account to look at the data locality problem in a global sense.

One of these parameters is the on-chip cache hierarchy, which determines the core connection and thus data sharing between computations on different cores. Another parameter is the memory controller. In a network-on-chip (NoC) based multicore architecture, an off-chip data access (main memory access) needs to travel through the on-chip network, spending considerable amount of time within the chip (in addition to the memory access latency). In addition, it contends with on-chip (cache) accesses as both use the same NoC resources. The third parameter that will be discussed in this thesis is the row-buffer. Many emerging multicores employ banked memory systems and each bank is attached a row-buffer that holds the most-recently accessed memory row (page). A last-level cache miss that also misses in the row-buffer can experience much higher latency than a cache miss that hits in the row-buffer. Consequently, optimizing for row-buffer locality can be as important as optimizing for cache locality.

Motivated by this, in this thesis, we propose four different compiler-directed “locality” optimization schemes that take these parameters into account. Specifically, our first scheme targets cache hierarchy-aware loop transformation strategy for multicore architectures. It determines a loop iteration-to-core mapping by
taking into account application data access pattern and multicore on-chip cache hierarchy. It employs “core vectors” to exploit data reuses at different layers of cache hierarchy based on their reuse distances, with the goal of maximizing data locality at each level while minimizing the data dependences across the cores. In case of dependence free loop nest, we customize our loop scheduling strategy, which, on the other hand, determines a schedule for the iterations assigned to each core, with the goal of reducing data reuse distances across the cores. Our experimental evaluation shows that the proposed loop transformation scheme reduces miss rates at all levels of caches and application execution time significantly, and when supported by scheduling, the reduction in cache miss rates and execution time become much larger.

The second scheme explores automatic data layout transformation targeting multithreaded applications running on multicores (which is also cache hierarchy-aware). Our transformation considers both data access patterns exhibited by different threads of a multithreaded application and the on-chip cache topology of the target multicore architecture. It automatically determines a customized memory layout for each target array to minimize potential cache conflicts across threads. Our experiments show that, our optimization brings significant benefits over state-of-the-art data locality optimization strategies when tested using 22 benchmark programs on an Intel multicore machine. The results also indicate that this strategy is able to scale to larger core counts and it performs better with increased data set sizes.

In the third scheme, focusing on multithreaded applications, we propose a compiler-guided off-chip data access localization strategy, which places data elements in the memory space such that an off-chip access traverses a minimum number of links (hops) to reach the controller that handles this access request. We present an extensive experimental evaluation of our compiler-guided optimization strategy using a set of 12 multithreaded application programs under both private and shared last level caches. The results collected emphasize the importance of optimizing the off-chip data accesses.

The fourth scheme presents a compiler-directed row-buffer locality optimization strategy. This strategy modifies the memory layout of data to increase the number of row-buffer hits without increasing the number of misses in the on-chip cache hierarchy. We implemented our proposed optimization strategy in an open-source compiler and tested its effectiveness in improving the row-buffer performance using a set of multithreaded applications.
Table of Contents

List of Figures viii
List of Tables xii

Chapter 1
Introduction 1
1.1 On-chip Cache Hierarchy ........................................... 2
1.1.1 Loop Transformation ........................................ 3
1.1.2 Data layout transformation .................................. 3
1.2 Memory Controller ................................................. 4
1.3 Row-Buffer ......................................................... 7

Chapter 2
Locality-Aware Loop Transformation for Multicores 10
2.1 Motivational Example ............................................ 10
2.2 Overview of Our Approach ...................................... 11
2.3 Program Representation .......................................... 13
2.4 Architectural Abstraction ........................................ 14
2.4.1 Core Vectors ................................................... 14
2.4.2 Virtual Cores .................................................. 16
2.5 Parallelization and Mapping .................................... 17
2.5.1 Iteration-to-Virtual Core Mapping ......................... 17
2.5.2 Virtual Core-to-Physical Core Mapping .................... 26
2.5.3 Code Generation .............................................. 27
2.6 Scheduling for Dependence-Free Loops ...................... 28
2.7 Experimental Evaluation ......................................... 30
Chapter 3
Optimizing Data Layouts for Parallel Computation on Multicores 36
3.1 Overview of Our Data Layout Transformation 36
3.2 Polyhedral Representation 39
3.2.1 Overview of Hyperplanes 39
3.2.2 Polyhedral Model 40
3.3 Hyperplane Transformation 41
3.4 Inter-thread Layout Optimization 44
3.4.1 Localization Test 44
3.4.2 Tiling Determination 46
3.4.3 Transformation Function Determination 47
3.4.4 Cache-Hierarchy Aware Memory Mapping 48
3.5 Experimental Evaluation 51
3.5.1 Setup 51
3.5.2 Results 56

Chapter 4
Off-Chip Data Locality Optimization 58
4.1 Motivational Results 58
4.2 Problem of Off-chip Accesses 60
4.3 Framework Overview 62
4.4 Off-chip Access Localization 63
4.4.1 Background 63
4.4.2 Determining Data-to-Core Mapping 65
4.4.3 Layout Customization 67
  4.4.3.1 Private L2s 68
  4.4.3.2 Shared L2 69
4.4.4 Extension to Page Interleaving 71
4.5 Experiments 73
  4.5.1 Experimental Setup and Applications 73
  4.5.2 Impact of Off-Chip Access Localization 75
  4.5.3 Evaluation of Cache-Block Interleaving 76
  4.5.4 Result with an Alternate L2-to-MC Mapping 76
  4.5.5 Sensitivity Analysis 77

Chapter 5
Row-buffer Locality Optimization 81
5.1 Background 81
5.1.1 Cache Basics 81
5.1.2 Row-buffer Basics 82
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2 Motivation and Overview</td>
<td>83</td>
</tr>
<tr>
<td>5.3 Row-buffer Locality-Aware Data Placement</td>
<td>85</td>
</tr>
<tr>
<td>5.3.1 Program Representation</td>
<td>85</td>
</tr>
<tr>
<td>5.3.2 Row-buffer Locality Test</td>
<td>86</td>
</tr>
<tr>
<td>5.3.3 Row-Buffer Locality Enhancement</td>
<td>88</td>
</tr>
<tr>
<td>5.3.3.1 Cache Miss Model</td>
<td>88</td>
</tr>
<tr>
<td>5.3.3.2 Identifying Memory Blocks</td>
<td>90</td>
</tr>
<tr>
<td>5.3.3.3 Reshuffling Memory Blocks</td>
<td>91</td>
</tr>
<tr>
<td>5.4 Multiple Arrays and Multiple Row-Buffers</td>
<td>94</td>
</tr>
<tr>
<td>5.5 Experimental Evaluation</td>
<td>98</td>
</tr>
<tr>
<td>5.5.1 Setup and Applications</td>
<td>98</td>
</tr>
<tr>
<td>5.5.2 Results</td>
<td>99</td>
</tr>
<tr>
<td>5.5.3 Summary of the Sensitivity Experiments</td>
<td>102</td>
</tr>
</tbody>
</table>

Chapter 6

Related Work 103

6.1 Loop Transformation 103
6.2 Data Transformation 104
6.3 Off-chip Data Localization 105
6.4 Row-buffer Locality 106

Chapter 7

Conclusion 108

Chapter 8

Future Work 110

Bibliography 115
List of Figures

1.1 Different types of multicore architectures with shared on-chip caches. Ovals represent cores and rectangles represent caches. In each figure, the cache closest to a core is L1. The numbers attached to the caches indicate their IDs. ..................................................... 2

1.2 Basic structure of a two-dimensional 4 × 4 NoC based multicore with 4 memory controllers (MCs). Each node hosts a processor core, private L1 data/instruction caches, an L2 cache (bank) which can be private (per core) and shared (across cores), and a router through which it gets connected to the neighboring nodes. Physical address space is distributed across memory controllers (MC1, MC2, MC3, and MC4) by the hardware. We assume L2 is the last level cache in the system. ................................................................. 5

1.3 (a) Flow of a memory access on an NoC based multicore with per core private L2 caches. The number attached to a node represents its core ID. (b) Flow of a memory access on an NoC based multicore with shared L2 caches [1]. The L2 cache is shared by all cores. ........................................... 5

1.4 Illustration of memory controller and row-buffers. ......................... 8

1.5 Normalized memory access latencies for two multithreaded applications. For each application, the last two bars are normalized respect to the first one. We observed a similar behavior with our other benchmarks as well. ................................. 9

2.1 An motivational example. .......................................................... 11

2.2 High level view of our approach. ............................................. 12

2.3 Division of data reuses into groups using separator vectors, for an s-layer cache hierarchy. .......................................................... 20

2.4 An illustration of iteration-to-physical core mapping for a two-level loop nest with a sample architecture. ................................. 26

2.5 Original code. ................................................................. 27

2.6 Blocked code. ................................................................. 27

2.7 Code generated by our scheme. ............................................. 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.8</td>
<td>Code generated by our scheduling scheme.</td>
</tr>
<tr>
<td>2.9</td>
<td>Illustration of our scheduling scheme (a) after the iteration-to-physical core mapping (b).</td>
</tr>
<tr>
<td>2.10</td>
<td>Improvements in cache misses and execution cycles.</td>
</tr>
<tr>
<td>2.11</td>
<td>Comparison against conventional optimization and the impact of scheduling.</td>
</tr>
<tr>
<td>2.12</td>
<td>Results compared to hierarchical tiling and polyhedral model.</td>
</tr>
<tr>
<td>2.13</td>
<td>Sensitivity to the selection of separator vectors.</td>
</tr>
<tr>
<td>3.1</td>
<td>A sample access pattern exhibited by 8 threads on a two-dimensional array.</td>
</tr>
<tr>
<td>3.2</td>
<td>Different data layouts for the localized array shown in Figure 3.1.</td>
</tr>
<tr>
<td>3.3</td>
<td>Band formed around a hyperplane (solid line) in the iteration space by reuse vectors (dashed lines) across the hyperplane.</td>
</tr>
<tr>
<td>3.4</td>
<td>Illustration of the distances (dashed lines) between array references (filled circles) and a hyperplane (solid line) in the data space.</td>
</tr>
<tr>
<td>3.5</td>
<td>Cache hierarchy-aware memory mapping for examples in Figure 1.1(c) and Figure 3.1.</td>
</tr>
<tr>
<td>3.6</td>
<td>L2 cache miss rates of our original benchmarks.</td>
</tr>
<tr>
<td>3.7</td>
<td>Execution times of our original benchmarks.</td>
</tr>
<tr>
<td>3.8</td>
<td>Reductions in L2 misses (default computation distribution).</td>
</tr>
<tr>
<td>3.9</td>
<td>Reductions in L2 misses (optimized computation distribution).</td>
</tr>
<tr>
<td>3.10</td>
<td>Reductions in execution times (default computation distribution).</td>
</tr>
<tr>
<td>3.11</td>
<td>Reductions in execution times (optimized computation distribution).</td>
</tr>
<tr>
<td>4.1</td>
<td>(a) Impact of off-chip data accesses. (b) Breakdown of the time spent by an off-chip access.</td>
</tr>
<tr>
<td>4.2</td>
<td>Virtual-to-physical address translation and the interpretation of the physical address bits.</td>
</tr>
<tr>
<td>4.3</td>
<td>Using data layout transformation to localize off-chip memory accesses.</td>
</tr>
<tr>
<td>4.4</td>
<td>Different L2-to-MC mappings. The data accessed by cores that belong to a cluster are allocated/accessed from the corresponding controllers (indicated by arrows).</td>
</tr>
<tr>
<td>4.5</td>
<td>The original and transformed data space for a set of data elements accessed by 64 cores.</td>
</tr>
<tr>
<td>4.6</td>
<td>The customized data layouts for the L2-to-MC mappings shown in Figures 4.4(b) and (a).</td>
</tr>
</tbody>
</table>
4.7 Changing Data-to-MC mapping through the OS support. In this example, we assume there are two memory controllers (MC1 and MC2).

4.8 Impact of off-chip access localization. The bars capture the accesses to MC1 in Figure 4.4a within an execution period. In the optimized case (b), most of accesses go to one memory controller (MC1), indicating that the off-chip accesses are localized.

4.9 The impact of localized accesses on application performance (page interleaving).

4.10 Application performance in the case that cache-block interleaving is adopted.

4.11 The execution time improvement results with the mappings shown in Figure 4.4(a) and (b).

4.12 Results with different controller placements.

4.13 Different MC placements with the same L2-to-MC mapping.

4.14 Configurations with 8 and 12 MCs.

4.15 Sensitivity to the number of MCs.

4.16 Sensitivity to the core count.

4.17 Application performance in the case that cache-block interleaving is adopted for SNUCA.

5.1 Motivational example. The black dots represent the data elements that incur misses in the last-level cache.

5.2 High-level view of our proposed approach.

5.3 (a) and (b) illustrate two different data footprints on the data space of array A, as well as the corresponding code fragments. The black dots represent the data elements, the solid black arrows indicate the direction of the data footprint, and the dashed boxes represent the memory blocks accessed by successive loop iterations.

5.4 An example that illustrates the concept of block distance.

5.5 Reshuffling the memory blocks to improve row-buffer locality for the example in Figure 5.3b.

5.6 Our strategy of reshuffling memory blocks without increasing the number of cache misses.

5.7 Target multicore based architecture.

5.8 Virtual-to-physical address translation and the use of the physical address bits in the target architecture.

5.9 The left portion is the original code and the right portion is the code after generating the initialization loop. Note that 256 in this loop is the page (row) size in terms of the number of data elements.
5.10 Reshuffling the memory blocks for the case of multiple row-buffers.
Now, instead of keeping a distance of $c$ (the case of single row-buffer), we keep the distance as $lcm(p, c)$. . . . . . . . . . . . . . . . . 96

5.11 Impact of data layout optimization. . . . . . . . . . . . . . . . . 100
5.12 Percentage reduction in execution cycles. . . . . . . . . . . . . . 100
5.13 Percentage reduction in execution cycles. . . . . . . . . . . . . . 100

8.1 (a) BLP and cycles-per-load in our applications. (b-d) BLP variation over time for gafort, apsi, and swim. . . . . . . . . . . . . . . . . 111
8.2 (a) An example of loop tiling. (b) The corresponding iteration space. (c) The corresponding data dependence graph (DDG). Arrows capture the data dependences among tiles. . . . . . . . . . . . . . . 112
8.3 The place of our proposed scheme in the compilation flow. . . . . . . . 112
List of Tables

2.1 Important characteristics of the Intel Dunnington architecture. . . 31
2.2 Benchmarks. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

3.1 Intel quad-core configuration. . . . . . . . . . . . . . . . . . . . . . 53
3.2 Benchmarks. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

4.1 The simulated configuration. . . . . . . . . . . . . . . . . . . . . . 73
4.2 Benchmarks used in our evaluation. . . . . . . . . . . . . . . . . . . 74

5.1 Experimental setup. . . . . . . . . . . . . . . . . . . . . . . . . . . 98
5.2 Benchmarks. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
Chapter 1

Introduction

The need to achieve high levels of performance without driving up power consumption and heating up computation fabric has become a major concern for chip vendors. Emerging multicore systems are being architected to improve performance and minimize heat/power consumption by integrating two or more cores into a single socket. While almost all vendors today already placed multicore architectures into their product lines, software – both application software and system software support – is lagging behind. This is unfortunate because – in parallel to these developments in architecture domain – many new application domains that can take advantage of multithreading are emerging. Unless software problem is handled in an appropriate and timely fashion, it may be difficult to reap up potential benefits brought by multicore systems. Therefore, the emergence of multicore platforms offers several opportunities for boosting application performance. These opportunities, which include parallelism and data locality benefits, require strong support from compilers as well as operating systems. However, architectural abstractions relevant to memory system are scarce in current programming and compiler systems. In fact, most compilers do not take any memory system specific parameter into account even when they are performing data locality optimizations. Instead, their locality optimizations are driven by rule-of-thumbs such as “maximizing stride-1 accesses in innermost loop positions”. There are a few compilers that take cache and memory specific parameters into account to look at
1.1 On-chip Cache Hierarchy

One of these parameters is the on-chip cache hierarchy, which determines the core connection and thus data sharing between computations on different cores. Figure 1.1 illustrates three sample multicore architectures whose cache topologies are quite different from one another. Consequently, a given multithreaded application can have very different performance characteristics in these architectures. One goal in this work is to automatically transform application programs for achieving better performance on on-chip cache hierarchies of emerging multicore architectures. We enlist help from an optimizing compiler to implement our automatic code optimization. Specifically, in this thesis, we propose two different cache-hierarchy aware data locality optimization schemes. While the first one optimizes the data locality through loop transformation, i.e., computation mapping and scheduling, the second one focuses on the data layout optimizations, which reorganize the data layout in the memory.
1.1.1 Loop Transformation

Let us now briefly explain how code mapping and code scheduling can help us achieve high degrees of constructive sharing and avoid destructive interferences. Basically, code mapping assigns each loop iteration to a core, which achieves a parallelization of a loop nest at the same time. If two loop iterations share data, it is better (from a constructive sharing perspective) to map them to cores that share a cache in some layer in the on-chip hierarchy. Further, the more intense the sharing is (i.e., more frequently it occurs), the more important to exploit the resulting reuse at higher layers of the cache hierarchy. On the other hand, if two iterations do not share data, it is better (from a destructive interference perspective) to assign them to cores that do not share any cache in the system (so that they do not displace each other’s data). If this is not done, they can conflict in shared caches. While careful mapping of loop iterations to cores can improve chances for constructive sharing and minimize chances for destructive interferences, a mapping can only dictate a placement of iterations (computations); it does not impose any order in execution of the iterations assigned to each core. Thus, mapping cannot guarantee that the data involved in the reuse will definitely be caught in that shared cache when the reuse actually takes place. This is where scheduling comes into the picture. Specifically, our goal in scheduling is to ensure that, if two iterations mapped to two different cores access the same data element (or data block), it is better that they access that data element/block in close proximity in time so that the second use (reuse) can catch the data while it is still in shared cache. Therefore, if applied carefully, mapping and scheduling can cooperatively maximize constructive sharing and minimize destructive interferences.

1.1.2 Data layout transformation

In addition to the computation mapping and scheduling scheme we discussed so far, another (complementary) optimization approach frequently used to improve the data locality is data transformations \[2\] \[3\] \[4\] \[5\], which change the layout of data in the memory space, and can be used to optimize data locality in cases where data dependences prevent loop restructuring and other types of code optimizations. However, most of these techniques are developed in the context of
single-core architectures. These localized data referenced by a single thread are usually distributed in the memory space when using conventional linear data layouts (such as row/column/diagonal major), even though they are geometrically close to each other in the data space of a multi-dimensional array. Consequently, accesses to such localized data (in random memory locations) by different threads create conflicts in shared on-chip caches, resulting in a situation which is similar to the destructive interferences experienced by multiprogrammed workloads, where different applications displace each other’s data from shared caches [6] [7] [8]. These inter-thread conflicts in shared caches can reduce the effectiveness of cache utilization and eventually degrade application performance. Further, since a cache line can hold multiple data elements, spreading the local data of a thread in the linear memory space increases memory footprint of the thread, which can also affect application performance.

Based on this observation, we explore a novel data layout optimization strategy, which (i) considers data access patterns exhibited by different threads of a multithreaded application and (ii) takes into account the on-chip cache hierarchy of the target multicore architecture. More specifically, our proposed approach identifies localized arrays in a parallel loop nest, applies data layout tiling to each of them, where the array is partitioned into disjoint portions such that each portion is accessed mostly by one thread, and then maps those portions (tiles) to the linear memory space in a cache hierarchy-aware fashion so that the working sets of parallel threads on each localized array are minimized at any given time frame during application execution. This helps not only reduce the cache conflicts across threads, but also decreases capacity misses in shared caches, thereby increasing the effectiveness of cache utilization.

1.2 Memory Controller

Another parameter is the memory controller. As shown in Figure 1.2, a state-of-the-art NoC based multicore (network-on-chip [1]) is constructed from multiple point-to-point data links connected by switches such that messages can be relayed from any source node to any destination node over several links by making routing decisions at the switches. Such a design brings several advantages from the scal-
Figure 1.2: Basic structure of a two-dimensional $4 \times 4$ NoC based multicore with 4 memory controllers (MCs). Each node hosts a processor core, private L1 data/instruction caches, an L2 cache (bank) which can be private (per core) and shared (across cores), and a router through which it gets connected to the neighboring nodes. Physical address space is distributed across memory controllers (MC1, MC2, MC3, and MC4) by the hardware. We assume L2 is the last level cache in the system.

Figure 1.3: (a) Flow of a memory access on an NoC based multicore with per core private L2 caches. The number attached to a node represents its core ID. (b) Flow of a memory access on an NoC based multicore with shared L2 caches [1]. The L2 cache is shared by all cores.

ability, complexity, power and reliability perspectives compared to conventional buses [1]. Optimizing data accesses in NoC based multicore systems has received considerable attention lately. The proposed strategies include careful design of cache access/lookup strategies [9, 10, 11] and on-chip access localization [12, 13].
While these optimizations are certainly important and can provide significant performance and power benefits, they are mostly oriented toward minimizing the number of cache misses and do not have much impact on off-chip accesses (main memory accesses), which can also be very important due to the following reasons:

- Since off-chip accesses must travel through the on-chip network to reach their target memory controllers, they can spend significant amount of time in the NoC, depending on the network congestion as well as the distance between the node that makes the off-chip request and the memory controller.

- Since off-chip accesses and on-chip accesses (cache accesses) share the same on-chip network, they contend for the same links and routers/buffers. Consequently, off-chip accesses also cause additional delays for on-chip cache accesses, further affecting the application performance.

Figures 1.3a and 1.3b show the memory access flows on an NoC based multicore with private (per core) or shared L2s, respectively. In the case of private L2s, when an L1 miss is detected, a request is sent to the local L2 cache. In case of an L2 miss, this request is forwarded to a centralized L2 tag directory, which is cached in the memory controller (MC) to which the requested data is mapped (path 1). The directory decides whether to get the data from another on-chip L2 cache (in which case we have an on-chip data access), or issue an off-chip memory request. In the latter case, the memory controller schedules this request (path 2), and the response from the corresponding memory bank is sent to the private L2 cache (path 3). In comparison, for the shared L2 case, based on the physical address, a given data block is assigned to an L2 bank (called the home bank) where it resides (when it is on-chip); and all off-chip requests for that data block are issued by its home (L2) bank. In order to look-up the corresponding L2 bank, a request is sent from L1 to L2 (path 1). If the data is a hit in the L2, then the response data is sent back to the L1 (path 5). In case of an L2 miss, the request is forwarded to the memory controller (path 2). The memory controller schedules this request (path 3) and the response of the memory module is sent first to the L2 (path 4) and then to the L1 (path 5).

The cost of an off-chip access in an NoC-based system is mainly a function of two parameters: (1) the time spent in the network (paths 1 and 3 in Figure 1.3a and paths 2 and 4 in Figure 1.3b), and (2) the time spent in memory controller and in
accessing the memory bank (path 2 in Figure 1.3a and path 3 in Figure 1.3b). In fact, our experiments show that, in an $8 \times 8$ multicore, cumulative network latency can be comparable to the memory access latency, which means the network latency can play a significant role in off-chip accesses. This latency can be reduced by improving the locality of off-chip accesses. What we mean by “improving locality” in this context is “reducing the distance” between the requester of an off-chip data element and the memory controller that manages the block/page that holds the data. For example, in Figure 1.3a, if the off-chip access request from the highlighted L2 bank is sent to MC3, the distance between the requester of the data and the node that is connected to the memory controller that holds the data is 10 links, whereas the distance for the off-chip access through path 1 is only 4 links. Everything else being equal, this L2 would prefer to satisfy most of its off-chip data requests from the nearest memory controller which is MC2.

1.3 Row-Buffer

The last memory parameter that will be discussed in this thesis is the row-buffer. Modern DRAM architectures are typically organized into multiple memory controllers, and each memory controller manages multiple banks, and each bank employs a buffer (called a row-buffer) that holds the most recently-accessed memory row (see Figure 1.4). Hitting in this row-buffer typically incurs 15 ns in DDR3-1333 whereas missing in it costs about 45 ns. Consequently, improving row-buffer hit rates can also play a critical role in shaping memory performance of an application. Recognizing this, several prior architecture oriented studies [14, 15, 16, 17, 18, 19, 20, 21, 22] targeted improving row-buffer hit rates/memory system performance and proposed different optimization schemes. One popular technique, called memory scheduling, reshuffles memory requests in the memory controller before they are issued to DRAM to maximize row-buffer hits, i.e., it prioritizes accesses destined to the most recently accessed row. Also, a prior compiler based study [23] explored the potential benefits of exploiting row-buffers in the context of single-threaded embedded applications and architectures with a single

---

1The impact of (2) can be reduced through smart memory controller scheduling algorithms as proposed in [14], which is orthogonal to our work.
Figure 1.4: Illustration of memory controller and row-buffers.

memory controller.

Emerging multicore systems however employ multiple memory controllers and run multiple threads, making it very difficult to achieve good row-buffer locality as interleaving accesses from different cores/threads minimizes chances for reusing the contents of a row-buffer. To illustrate the impact of this on application performance, we present in Figure 1.5 normalized average data access latency for two multithreaded application programs. For each application, we have three bars: the first one represents the performance from the execution on a 32-core system, with one thread per core; the second represents the case where all off-chip accesses (last level cache misses) miss in the row-buffer; and the last one shows the same when all off-chip accesses hit in the row-buffer. As can be observed from these plots, missing in the row-buffer versus hitting in it can make a significant difference. And, both of these multithreaded applications miss most of their last-level cache misses in the row-buffer as well (the first bar is much closer to the second bar than to the third bar). These results clearly motivate for optimizing row-buffer performance, which is the focus of this paper.

However, optimizing for row-buffer performance is not trivial due to multiple factors. First, one cannot simply treat a row-buffer as another layer in the memory/cache hierarchy since it has only one (very large) line (row) and any access that falls outside it will cause a row-buffer replacement destroying any locality. For example, when accessing the elements of, say, two arrays in a loop, each access will replace the contents of the row-buffer (even though each access can exhibit
Figure 1.5: Normalized memory access latencies for two multithreaded applications. For each application, the last two bars are normalized respect to the first one. We observed a similar behavior with our other benchmarks as well.

spatial locality when considered alone). Second, any code or data restructuring for optimizing row-buffer performance has the potential of negatively impacting cache performance. This issue is particularly problematic for programs that have already been optimized for the best cache performance. In other words, any code or data restructuring that target improving row-buffer hits should not affect cache hits rates if at all possible. Third, trying to optimize row-buffer accesses means dealing with cache misses (rather than data accesses), and is consequently less deterministic as the compiler cannot in general predict which data accesses will miss in the cache very accurately. Motivated by this, our goal is to increase the number of row-buffer hits of multithreaded applications running on multicores without increasing cache misses. To achieve this, we propose a novel compiler-based row-buffer locality optimization strategy.

The rest of this thesis is organized as follows. Chapter 2 to Chapter 5 present the technique details of our proposed schemes, Chapter 6 discusses the related work and we conclude in Chapter 7 by summarizing the contributions of this thesis. Chapter 8 will point out future research directions.
2.1 Motivational Example

Tiling/blocking is a primary loop transformation techniques in improving cache performance of scientific programs. It partitions iterations space of a loop nest into smaller chunks (blocks), the therefore changes the iteration execution order to ensure that data reuses can be satisfied from cache memory. In static scheduling domain, most of the existing tiling techniques distribute tiles across physical cores in a block-cyclic fashion. Figure 2.1(b) and (c) give two examples. Note that, in Figure 2.1(b), tiles are distributed along a single dimension, while in Figure 2.1(b), tiles are distributed along two dimensions. Generally speaking, the example in Figure 2.1(c) will result fewer communications among the cores, but lower degree of parallelism. In comparison, Figure 2.1(c) will have more communications among the cores, but higher degree of parallelism. In this work, we only focus on the latter case, i.e., multi-dimensional blocking. A common criteria to measure the goodness of a tiling scheme is the communication volume between tiles since after distributing tiles among physical cores, these volume determines the communications among physical cores. However, only considering communication volume may not be enough in the context of multicores. Next, we show that data locality also plays an important role in determining a good tiling scheme. To explain this,
Figure 2.1: An motivational example.

Let us assume that in Figure 2.1(b), i-loop is the innermost loop. When the upper loop bound of the innermost loop is very large, most of the accesses in the horizontal boundaries (gray area) will incur misses in different level of caches. Even the communication volume among tiles are low, those misses can still degrade the overall application performance. Now let us consider another tiling scheme shown in Figure 2.1(d). In this case, the accesses in the horizontal boundaries may have incur hits in the caches if we consider the architecture in Figure 2.1(a).

### 2.2 Overview of Our Approach

Figure 8.3 illustrates the high-level view of our proposed compiler-based approach. For each loop nest, we apply our cache topology-aware parallelization and locality optimization scheme, which consists of two steps, namely, *mapping* and *scheduling*. Mapping determines how loop iterations are distributed across the cores, while scheduling decides the execution order of loop iterations assigned to each core. Both these steps take into account intra-core and inter-core data access and reuse patterns as well as the on-chip cache hierarchy of the target multicore architecture.

In particular, the mapping component further contains two sub-steps: *iteration-to-virtual core mapping* and *virtual core-to-physical core mapping*. The former
one employs a concept of virtual cores to represent a “virtual architecture” that has the same number of levels of caches as the target (physical) architecture, but infinite number of cores and cache components at each level, and determines an iteration-to-core mapping such that data reuses with different reuse distances (represented by reuse vectors in our work) can be exploited at different layers of the cache hierarchy. The core idea behind this step is that, data reuses with shorter reuse distances should be exploited in the higher level caches that have faster accesses (i.e., ones that are closer to cores), since they occur more frequently during program execution and usually take place in a shorter period of time, whereas data reuses with longer reuse distances can be exploited in the lower levels of caches that have larger capacities. In other words, data reuses with different reuse distances are expected to be converted into locality at different layers of the cache hierarchy that have different characteristics (in terms of access latency and capacity). Note that, this can only be achieved through a cache topology-aware computation mapping. Once the iteration-to-virtual core mapping is done, the second sub-step employs a folding function to map infinite virtual cores to limited physical cores by considering the data dependences across the cores. As a result, a loop nest is parallelized and all its iterations are properly assigned to cores. Note that, our mapping strategy also tries to maximize the utilization of all (physical) cores available; that is, it is oriented toward maximizing data locality as well as parallelism.

After computation-to-core mapping, the scheduling component takes the iterations assigned to physical cores and determines an execution order for them, with two main goals: (i) satisfying data dependences within each core and across the cores, and (ii) tuning the timing of data sharing (reuse) among the cores so that
data reuse can be converted into locality during program execution. It is worth
mentioning that, both our mapping and scheduling strategies are formulated and
implemented in a linear algebraic framework; therefore, they can easily be in-
tegrated with many previously-proposed high-level parallelism and data locality
optimizations.

2.3 Program Representation

Our work targets loop and data intensive applications. In these applications, an
l-level loop nest represents an l-dimensional iteration space, and each point in this
space (i.e., an iteration) can be denoted by an iteration vector \( \vec{i} = (i_0, i_1, \ldots, i_{l-2}, i_{l-1})^T \),
where \( i_j (0 \leq j \leq l) \) can take values from \([L_j, U_j]\), \( L_j \) and \( U_j \) being, respectively,
the lower and upper bounds of the \( j \)th loop from the outermost. The original
execution order of loop iterations obeys lexicographical ordering\(^1\) (denoted using
\( \prec \)). The iteration space of a loop nest can also be viewed as a bounded polyhedron
[24]. If two iterations \( \vec{i}_1 \) and \( \vec{i}_2 \) access the same data element, we say that these two
iterations exhibit temporal data reuse [25]. Similarly, if they access data elements
that are mapped to the same cache line (block) boundary in the memory space,
they are said to have spatial data reuse [25]. In either case, assuming that \( \vec{i}_1 \prec \vec{i}_2 \),
\( \vec{r} = \vec{i}_2 - \vec{i}_1 \) is called the reuse vector [25], which represents the reuse distance be-
tween two accesses to the same data element. Moreover, if at least one of these
two iterations is a write operation to the data element, this reuse vector is also
called a dependence vector [26]. Note that a dependence vector can be seen as a
special type of reuse vector, which enforces a specific execution order between the
two iterations involved.

Reuse vectors can be analyzed and calculated based on pairs of array index ex-
pressions. In the case that two references to an array are uniformly generated; that
is, references whose array index expressions differ in at most a constant term, e.g.,
\( A[i, j+1] \) and \( A[i, j] \), the reuse vector is constant, e.g., \((0, 1)^T\). However, in the case
that two references to an array are non-uniformly generated, e.g., \( A[i + j, j] \) and
\(^{1}\)Let \( \vec{x} = (x_1, x_2, \ldots, x_n) \) and \( \vec{y} = (y_1, y_2, \ldots, y_n) \) be two vectors in a n-dimensional space. \( \vec{x} \) is
said to be lexicographically smaller than \( \vec{y} \), denoted by \( \vec{x} \prec \vec{y} \) if \( x_1 < y_1 \) or both \( x_1 = y_1 \) and
(\( x_2, \ldots, x_n \prec y_2, \ldots, y_n \)).
A[i, j], the reuse vector is a symbolic expression containing loop iterator variables, e.g., (j, 0)^T. Note that, a reuse vector computed based on array index expressions usually denotes the reuse distance(s) for a set of (multiple) loop iteration pairs, irrespective of whether the reuse vector is constant or not, since array index expressions are expressed in terms of loop iterator variables (i.e., i_j, where 0 ≤ j ≤ l) bounded by loop bounds, and as loop iterators vary, different loop iterations can be enumerated. Consequently, each reuse vector is in fact implicitly associated with a domain that specifies the loop iterations involved in the reuses represented by this vector. For example, if 0 ≤ i ≤ 1 and 0 ≤ j ≤ 1, the domain of the constant reuse vector (0, 1)^T generated from references A[i, j + 1] and A[i, j] and the domain of the non-constant reuse vector (j, 0)^T generated from references A[i + j, j] and A[i, j] are \{(0 0)^T, (0 1)^T, (1 0)^T, (1 1)^T\}.

Reuse vector is the primary abstraction used for data locality optimization. For a given loop nest, we use \( R = (\vec{r}_0, \vec{r}_1, ..., \vec{r}_{q-1}) \), called the reuse matrix, to represent all its reuse vectors, where each column \( \vec{r}_i \) is a reuse vector identified in this loop nest. Clearly, from a performance point of view, we prefer small reuse vectors over large ones (in a lexicographic sense). This is because small reuse vectors indicate shorter reuse distances, which means we have higher chances for catching the reused data in the cache.

2.4 Architectural Abstraction

2.4.1 Core Vectors

The primary architectural abstraction used by our compiler-based approach is built on a novel concept called core vector. In a multicore architecture with multiple levels of caches, each core can be represented (tagged) using a core vector. To assign core vectors to cores, we first enumerate all the caches and cores from left to right at each layer of the hierarchy, as shown in Figure 1.1, and then represent each core using a vector that is composed of the core ID followed by IDs of all the caches it accesses in a top-down order (i.e., from the fastest/smallest cache to the slowest/largest). For example, in Figure 1.1(c), the core vectors for core 0 and core 1 are \( \vec{c}_0 = (0, 0, 0, 0)^T \) and \( \vec{c}_1 = (1, 1, 0, 0)^T \), respectively. If an architecture
has $s$ levels in its cache hierarchy (including cores as a separate layer), the core vector for core $k$ ($0 \leq k < n$), where $n$ is the total number of cores, would be $\mathbf{c}_k = (c_{k_0}, c_{k_1}, ..., c_{k_{s-1}})^T$, where $c_{k_0}$ is the ID of core $k$ and $c_{k_1},...,c_{k_{s-1}}$ are IDs of the caches that core $k$ accesses at each level. Clearly, this numbering strategy can easily be extended to architectures with larger number of cores and cache layers.

The motivation behind the use of core vectors is that, when two iterations (or threads or computations) are assigned to two cores with core vectors $\mathbf{c}_i$ and $\mathbf{c}_j$ ($0 \leq i, j < n$), respectively, by comparing the two core vectors, one can determine how these two iterations share cores and caches at different layers of a hierarchy, and therefore further decide whether the data reuse between these two iterations could potentially be exploited in the cache hierarchy, and if so, at what level.

For example, if two iterations are mapped to two cores $c_0$ and $c_1$ in Figure 1.1(c) with core vectors $(0, 0, 0, 0)^T$ and $(1, 1, 0, 0)^T$, the data reuse between these two iterations can be exploited in the L2 and L3 caches connected to these two cores, since the two core vectors have the same L2 and L3 IDs. As another example, if two iterations are mapped to two cores $c_0$ and $c_4$ that have core vectors $(0, 0, 0, 0)^T$ and $(4, 4, 2, 1)^T$, the data reuse between these two iterations cannot be exploited in any cache in the hierarchy, since the two core vectors have no common entries, which means the corresponding two cores do not share cache at all. We now formally define the core difference vector to capture the sharing patterns of computations (iterations) mapped to two cores.

**Definition 2.4.1.** Let $\mathbf{c}_i$ and $\mathbf{c}_j$, $0 \leq i, j < n$, $\mathbf{c}_i, \mathbf{c}_j \in \mathbb{Z}^s$, be two core vectors, where $n$ is the total number of cores, and $s$ is the total number of levels in a cache hierarchy (including cores). The core difference vector $\mathbf{d}_{i,j}$ is defined as $\mathbf{d}_{i,j} = (d_0, d_1, ..., d_{s-1})^T$, where $d_t$ ($0 \leq t < s$) is calculated as:

$$d_t = \begin{cases} 
0, & \text{if } c_{i,t} = c_{j,t}; \\
1, & \text{otherwise.}
\end{cases}$$

Note that, cache hierarchy in the multicore architectures is usually a “tree-like” structure, where the shared off-chip memory can be seen as the root, caches as internal nodes, and cores as leaves. This means that, if cores $i$ and $j$ share a cache at level $t$ ($1 \leq t < s$), they will share caches at levels $t + 1, t + 2, \cdots, s - 1$.
as well; consequently, after the first occurrence of a zero value, say, \( d_t = 0 \) is detected, all the values from \( d_{t+1} \) to \( d_{s-1} \) will be 0 in the core difference vector \( \vec{d}_{i,j} \). However, our model is general enough to capture other (possibly, asymmetric) cache connections as well.

### 2.4.2 Virtual Cores

Given a target architecture with \( s \) levels of hierarchy for cores and caches, all the core vectors collectively define an \( s \)-dimensional core space, where the core vectors correspond to some discrete points in the space, e.g., \((0, 0, 0, 0)^T\), \((1, 1, 0, 0)^T\), and \((2, 2, 1, 0)^T\) for the architecture shown in Figure 1.1(c). However, other core vectors such as \((0, 1, 0, 0)^T\) are not legal, since they do not have any corresponding points in the core space. Therefore, the core space for a real multicore architecture with a “tree-like” on-chip cache hierarchy is not continuous.

To determine a mapping for all the iterations in a loop nest, our strategy is to first construct a continuous linear core space that consists of virtual core vectors (where each virtual core vector represents a virtual core), then use a mapping matrix \( M \) to map each loop iteration to a virtual core vector in this space,\(^2\) and finally employ a folding function to map multiple virtual core vectors to fewer, legal virtual core vectors that correspond to physical cores in the target architecture. The virtual cores represent a virtual architecture which has the same number of levels of cache hierarchy as the target architecture; however, it has an infinite number of cores and caches at all levels, and it allows a core or a cache at level \( t \) \((0 \leq t < s)\) to be connected with any cache at level \( t + 1 \) \((0 \leq t < s - 1)\). As a result, a virtual architecture has a general “graph-like” structure; and any “tree-like” on-chip hierarchy can be considered as a special case of this virtual architecture. In other words, a virtual architecture is a large template onto which many physical architectures with the same number of cache layers can be mapped.

Virtual cores serve two purposes in our approach. Firstly, they provide a continuous linear space for applying parallelization and mapping, which makes our approach easier to formulate. If virtual cores were not there, in our mathematical formulations (explained later), we would have to deal with constant bounds (e.g.,

\[^2\]Note that, this requires that \( \forall \vec{i}, M. \vec{i} \geq 0 \).
number of cores), which would make implementation harder. Virtual cores help us separate the linear algebraic analysis from the mapping problem. Secondly, once we determine our mapping function for a virtual architecture (i.e., in terms of virtual core vectors), we can use them for any physical architecture that fits the template and customize only the folding function for each physical architecture. Furthermore, although not explored in this study, in cases where the target cache topology is not completely known at compile time, the code generated by our approach can be customized at run-time using the folding function.

### 2.5 Parallelization and Mapping

#### 2.5.1 Iteration-to-Virtual Core Mapping

Given a target cache hierarchy (including cores), this step employs a mapping matrix $M$ to map each iteration of a loop nest to a virtual core, such that existing data reuses can be exploited at different layers of cache hierarchy based on their reuse distances.

Recall that data reuse can be represented by reuse vectors in a loop nest, as explained in Section 2.3. If two iterations $\vec{i}_1$ and $\vec{i}_2$ exhibit a data reuse between them with a reuse vector of $\vec{r} = \vec{i}_2 - \vec{i}_1$ (assuming $\vec{i}_2$ is lexicographically greater than $\vec{i}_1$) and we want to exploit their data sharing at level $k$ ($0 < k < s$) in the cache hierarchy of a target (physical) architecture, then we should map these two iterations to two physical cores with core vectors $\vec{c}_i$ and $\vec{c}_j$, whose core difference vector $\vec{d}_{i,j}$ has the following format:

$$d_{i,j} = \begin{pmatrix} s \text{ entries} \\ \hline k \text{ entries} \\ \hline \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ \cdots \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}^T.$$

That is, the two physical cores should share some cache at level $k$ (and below),\(^3\) so that the data reuse between these two iterations has a chance to get converted into locality at level $k$. Note however that this does not guarantee that the reuse

\(^{3}\)We assume that caches are inclusive and they have a “tree-like” structure in the target architecture, which means that, if a data element is found in a cache at level $k$, it can also be found in the cache that connects to this cache at level $k + 1$. 

targeted will certainly be exploited at level $k$ or the data element in question will
definitely be caught at level $k$ at the time of reuse. It depends on the times at
which these iterations ($\vec{i}_1$ and $\vec{i}_2$) are executed, i.e., on their scheduling. Mapping
only ensures that $\vec{i}_1$ and $\vec{i}_2$ with data reuse are mapped to cores that share a cache
between them, and it creates an opportunity for exploiting the reuse at level $k$.

Further, since $d_{i,j} = f(\vec{i}_2) - f(\vec{i}_1)$, we have $d_{i,j} = M.\vec{i}_2 - M.\vec{i}_1 = M.(\vec{i}_2 - \vec{i}_1) = M.\vec{r}$. The expression above can be re-written as:

$$M.\vec{r} = \begin{pmatrix} 1 & 1 & \cdots & 1 & 0 & 0 & 0 \end{pmatrix}^T.$$  
(2.1)

This tells us that, we can use reuse vectors of a loop nest to decide the mapping
matrix $M$.

Note that, as mentioned before, we would like to exploit data reuses with
shorter reuse distances (represented by shorter reuse vectors) at higher layers of the
cache hierarchy since they occur more frequently and hence take better advantage
of low access latency provided by the caches at higher layers. We also prefer
to exploit data reuses with longer reuse distances (represented by longer reuse
vectors) at lower layers of caches since such reuses can be caught in the caches
with larger capacities when taking place. Therefore, given $q$ reuse vectors in a
loop nest, denoted by the reuse matrix $R = (\vec{r}_0, \vec{r}_1, ..., \vec{r}_{q-1})$, we first sort them in
a lexicographically-ascending order based on their absolute values, e.g., $\vec{r}_0 \preceq \vec{r}_1 \preceq \cdots \preceq \vec{r}_{q-1}$ (since the absolute value of a reuse vector represents its reuse distance),
then divide them into groups denoted by $g_i$ ($1 \leq i < s$) according to the layers of
cache hierarchy using separator vectors $\vec{\delta}_i$ ($1 \leq i < s - 1$), and finally map them
to different cache layers using Eq.(2.1) by determining a proper mapping matrix
$M = (\vec{m}_0 \ \vec{m}_1 \ \vec{m}_2 \ \cdots \ \vec{m}_{s-1})^T$, where $\vec{m}_i$ ($0 \leq i < s$) is a row vector of
$M$. This idea is pictorially illustrated in Figure 2.3 for an $s$-layer cache hierarchy,
where all the reuses lexicographically smaller than or equal to $\vec{\delta}_1$ are expected to
be exploited at the L1 cache layer, and all the reuses that are larger than $\vec{\delta}_1$ and
smaller than or equal to $\vec{\delta}_2$ are expected to be exploited at the L2 cache layer, and
so on, so that the performance of the on-chip cache hierarchy can be maximized.
The separator vectors $\vec{\delta}_i$ ($1 \leq i < s - 1$) are selected based on the cache capacity
at each layer, as will be discussed shortly. It is to be noted at this point that, we do not show $\vec{m}_0$ in Figure 2.3, which corresponds to the core layer (i.e., layer 0) in the hierarchy, since we decide it by using a group of dependence vectors that are chosen from all available reuse vectors (recall that dependence vector is a special type of reuse vector), with the goal of minimizing synchronization overhead across the cores.

In this case, Eq.(2.1) can be written as:

$$M. \vec{d} = \begin{pmatrix} \text{s entries} \\ 0 & 0 & \cdots & 0 \end{pmatrix},$$

(2.2)

where $d$ is a dependence vector. To summarize, our general approach in the “parallelization and mapping” step is to determine a mapping matrix $M$ based on the cache hierarchy and a classification of reuse vectors, and then use this matrix to map all iterations to cores.

Observe that, Eq.(2.1) and Eq.(2.2) represent an iteration-to-physical core mapping, since their right-hand side is the format for a physical core difference vector $d_{i,j}$. Recall that, cache hierarchy in real multicore architectures is usually a “tree-like” structure, and if the $k^{th}$ ($0 \leq k < s$) entry in $d_{i,j}$ is 0, all the entries following it are 0. However, the virtual architecture constructed based on a given target (physical) architecture allows arbitrary connections between adjacent layers in the hierarchy, which means that, in exploiting the data sharing (or data dependence) between two iterations at layer $k$, where $0 \leq k < s$, we only need to care about the $k^{th}$ entry in their virtual core difference vector and ensure that it is 0. Therefore, we have the following equation for iteration-to-virtual core mapping:

$$M. \vec{u} = \begin{pmatrix} \text{s entries} \\ X \cdots X & 0 & X \cdots X \end{pmatrix},$$

(2.3)

where $\vec{u}$ denotes either a reuse vector $\vec{r}$ or a data dependence vector $\vec{d}$.

In addition, since $M = (\vec{m}_0 \; \vec{m}_1 \; \vec{m}_2 \; \cdots \; \vec{m}_{s-1})^T$, we have $\vec{m}_0 \cdot \vec{d} = 0$ if we want to map the two iterations involved in certain data dependence represented by the dependence vector $\vec{d}$ onto the same core, and $\vec{m}_k \cdot \vec{r} = 0$ ($0 < k < s$) if we want to exploit certain data reuse represented by the reuse vector $\vec{r}$ between two
iterations in a shared cache at level $k$, e.g., $\vec{r} \in g_k$.

It can be noticed that, this allows us to solve each row vector in matrix $M$ independently and makes our mathematical formulation simpler. Note also that, we still need to obtain a mapping between iterations and physical cores ultimately, which should conform to the format of the core difference vector in Eq.(2.1) and Eq.(2.2). Although our iteration-to-virtual core mapping step does not guarantee this, our virtual core-to-physical core mapping will ensure it through a folding process, as discussed in Section 2.5.2.

In the following discussion, we present the technical details of how to determine a mapping matrix $M$ for an $l$-level loop nest and an $s$-level cache hierarchy. In order to achieve a one-to-one/onto (many-to-one) mapping from the iteration space to the virtual core space, we only need $\min(l, s)$ number of row vectors in $M$, as proved by the following lemma. Note that, to have a valid mapping, the mapping function should be either one-to-one or onto.

**Lemma 2.5.1.** Given a cache hierarchy of $s$ layers (including cores) and an $l$-dimensional loop nest, to achieve a one-to-one or onto iteration-to-virtual core mapping, only $\min(l, s)$ number of row vectors are required in matrix $M$.

**Proof:** First, the number of columns in $M$ is $l$, since our iteration-to-virtual core mapping is determined by $M \vec{i}$, and $\vec{i}$ is an $l$-dimensional vector. Second, in the case that $s < l$, we can always find $s$ linearly independent row vectors to construct an $s \times l$ mapping matrix $M$. Let the linear transformation $T : Z^l \to Z^s$ be defined as $T(\vec{i}) = M \vec{i}$. Since the rows of $M$ are linear independent, the columns of $M$ span $Z^s$; therefore, $T$ is an onto (many-to-one) transformation (mapping). In the case that $s > l$, we can find $l$ linearly independent row vectors to construct an $M$, and the columns of $M$ are also linearly independent, which implies that $T$
is a one-to-one mapping. Therefore, we can always use \( \min(l, s) \) number of row vectors to construct matrix \( M \).

Our approach to determining a mapping matrix \( M \) includes two steps: (i) reuse vector grouping, and (ii) row vector determination.

**Step 1: Reuse Vector Grouping**

The first step is to divide all the reuse vectors in the loop nest into groups \( (g_i, 1 \leq i < s', \text{where } s' = \min(l, s)) \) based on the target cache hierarchy and cache capacities (In the case that \( s' < s \), we use only \( s' \) top levels of the hierarchy to decide the grouping). Suppose that, we have \( q \) reuse vectors, represented by the reuse matrix \( R = (\vec{r}_0, \vec{r}_1, \ldots, \vec{r}_{q-1}) \), and each is associated with a domain that specifies the iteration pairs involved in its reuse. To group them, we first sort them in a lexicographically-ascending order based on their absolute values. Recall however that a reuse vector may be constant (uniform reuse) or non-constant (non-uniform reuse), and we cannot perform sorting directly if there are non-constant vectors. To solve this problem, we first enumerate all distinct values for each non-constant vector using its domain information (see Section 2.3), i.e., convert each non-constant vector into a set of constant vectors, and then sort them along with the constant vectors.

Let us assume that, after sorting, we have \( \vec{r}_0' \leq \vec{r}_1' \leq \cdots \leq \vec{r}_{p-1}' \), where \( \vec{r}_i' (0 \leq i < p) \) is either an original constant vector from \( R \) or a constant vector converted from a non-constant one, and \( p \) is the total number of (distinct) constant vectors.

Now, we need to select separator vectors in order to partition these reuse vectors into groups. We determine our separator vectors using the cache capacities at different layers of the cache hierarchy. Let \( C_k \) denote the capacity of a level-\( k \) cache, \( \vec{\delta}_k = (i_0 \ i_1 \ \cdots \ i_{l-1})^T \) represent the target separator vector for determining group \( k \), where \( 1 \leq k < s' - 1 \), as depicted in Figure 2.3, \( B_i \) (\( 0 \leq i < l \)) denote the length (i.e., total number of iterations) of the \( i^{th} \) loop in the loop nest,\(^4\) and \( E \) represent the size of all data elements accessed by one iteration. Then, the separator vector \( \vec{\delta}_k \) should satisfy the following equation:

\[
E \times (i_0 \times B_1 + i_1 \times B_2 + \cdots + i_{l-2} \times B_{l-1} + i_{l-1}) = C_k.
\]

\(^4\)In the case that a loop bound is symbolic, we use profiling to obtain its length.
Although this equation has many solutions, we take the lexicographically smallest solution as the value for $\vec{\delta}_k$. The meaning of $\vec{\delta}_k$ determined by this equation is that, the number of data elements accessed (estimated by $E \times (i_0 \times B_1 + i_1 \times B_2 + \cdots + i_{l-2} \times B_{l-1} + i_{l-1})$) between any two iterations involved in a data reuse represented by a reuse vector $\vec{r}$, which is smaller than $\vec{\delta}_k$, should be less than the size of a level-$k$ cache, so that, at the time of the reuse, the reused data element has a chance to be caught in that cache.

**Step 2: Row Vector Determination**

After grouping the reuse vectors, we next determine the row vectors for matrix $M$. Specifically, we divide them into two cases, $\vec{m}_0$ and $\vec{m}_k$ ($1 \leq k < s'$), which are slightly different from each other.

**Case 1. Determining $\vec{m}_0$.**

In this case, we select a set of dependence vectors to determine $\vec{m}_0$ based on Eq.(2.3), since we want the iterations involved in these data dependences to be mapped to the same core, which can help reduce the costs of inter-core synchronization. According to Eq.(2.3), we have $\vec{m}_0 . \vec{d} = 0$, where $\vec{d}$ is a dependence vector. Note that, to determine $\vec{m}_0$ in this equation, we can select at most $(l - 1)$ linearly independent dependence vectors. Since there might be more than $(l - 1)$ dependence vectors, our strategy is to select $(l - 1)$ lexicographically smallest ones from the sorted sequence $\vec{r}'_0 \preceq \vec{r}'_1 \preceq \cdots \preceq \vec{r}'_{p-1}$ obtained in the first step. Assuming that $\vec{d}_1, \vec{d}_2, \cdots, \vec{d}_{l-1}$ are the selected ones, then we have:

$$\vec{m}_0 . \vec{d}_1 = 0; \quad \vec{m}_0 . \vec{d}_2 = 0; \quad \cdots; \quad \vec{m}_0 . \vec{d}_{l-1} = 0. \quad (2.4)$$

For the remaining dependence vectors $\vec{d}$ that are not selected or used in Eq.(4.1), we should satisfy the following condition to ensure the legality of the transformation:

$$\vec{m}_0 . \vec{d} \geq 0. \quad (2.5)$$

Also, for all the reuse vectors $\vec{r}'$ that are not used in Eq.(4.1) (including the non-selected dependence vectors $\vec{d}'$), we have the following constraint:

$$|\vec{m}_0 . \vec{r}'| \leq q_0, \quad (2.6)$$
with the goal of minimizing $q_0$. The motivation for setting up this last constraint is that, even if the iterations involved in the reuse represented by the reuse vector $\vec{r}'$ cannot be mapped to the same virtual core ($|\vec{m}_0, \vec{r}'| \neq 0$), we still want to minimize the distance between the two (different) virtual cores they are mapped to, i.e., we want to minimize the virtual-core mapping distance $|\vec{m}_0, \vec{r}'|$. On the one hand, a smaller virtual-core mapping distance can lead to a smaller physical-core mapping distance (i.e., the distance between physical cores onto which the iterations involved in the reuse are mapped) because of our folding function (explained later); on the other hand, in real architectures, cache hierarchy is usually a “tree-like” structure, and the closer the two physical cores, the larger chance they share a cache at a higher level in the hierarchy, and therefore, mapping iterations to two nearby physical cores usually leads to better locality.

At this point, we have a system of constraints: Eq.(4.1), Eq.(4.2) and Eq.(4.3). We now employ the integer Fourier-Motzkin Elimination (FME)\(^5\) to find the minimal $q_0$, under which we select the lexicographically smallest vector as the solution for $\vec{m}_0$. Note that, there may be less than $(l - 1)$ dependence vectors, or no dependence vector at all in the loop nest, which means that we have fewer (or no) equalities in Eq.(4.1), in which case we use the integer FME to find a solution based on constraints Eq.(4.2) and Eq.(4.3), plus whatever equalities provided by Eq.(4.1).

If the integer FME does not return a non-trivial solution (i.e., return only the trivial solution of $\vec{0}$), we drop all the constraints involving the largest (in a lexicographic sense) reuse or dependence vector among all, and attempt to solve the resulting reduced system of constraints using the integer FME. If there is still no solution, we continue to drop the second largest reuse or dependence vector, and try to solve the remaining constraints, and so on, until a non-trivial solution is found.

\(^5\)Fourier Motzkin Elimination (FME) [24] is a mathematical algorithm for eliminating variables from a system of linear inequalities. Elimination of variables, $V$, from a system of relations (here, linear inequalities) involves creating another system of the same kind, but without the variables $V$, such that both systems have the same solutions over the remaining variables. Let us consider a system $S$ of $n$ inequalities with $r$ variables, $x_1$ through $x_r$, with $x_r$ being the variable which we want to eliminate. Each linear inequality that involves $x_r$ can be re-written as $\sum_{k=1}^{r-1} b_k x_k \leq x_r \leq \sum_{k=1}^{r-1} a_k x_k$, which is equivalent to $\sum_{k=1}^{r-1} b_k x_k \leq \sum_{k=1}^{r-1} a_k x_k$. This effectively eliminates $x_r$. 
Case 2. Determining $\vec{m}_k$ ($1 \leq k < s'$).

Similar to Case 1, to determine the row vector $\vec{m}_k$ for $M$ based on Eq.(2.3), we select at most $(l - 1)$ linearly independent reuse vectors from group $g_k$ ($1 \leq k < s'$), which are lexicographically smallest. Suppose that $\vec{r}_1', \vec{r}_2', \cdots, \vec{r}_{l-1}'$ are the selected reuse vectors. Then, we have:

$$\vec{m}_k.\vec{r}_1' = 0; \quad \vec{m}_k.\vec{r}_2' = 0; \quad \cdots \quad \vec{m}_k.\vec{r}_{l-1}' = 0.$$  \tag{2.7}

This means that, we want the iterations involved in these data reuses to be mapped to the same shared cache at layer $k$ in the hierarchy, in order to improve data locality. Note that, any reuse vector or dependence vector can be selected and used only once in the equalities captured by Eq.(4.1) and Eq.(2.7).

We also have the following constraint to ensure the legality of the transformation:

$$\vec{m}_k.\vec{d} \geq 0, \text{ for all dependence vectors } \vec{d} \in R,$$  \tag{2.8}

and the constraint to minimize the mapping distance at layer $k$ (with a similar reason for Case 1):

$$|\vec{m}_k.\vec{r}'| \leq q_k,$$  \tag{2.9}

for all the reuse vectors $\vec{r}' \in R$ that are not used in Eq.(2.7). At this point, we use the integer FME to solve Eq.(2.7), Eq.(2.8) and Eq.(2.9) by finding the minimal $q_k$. Note also that $q_k$ may be or may not be the same as the $q_t$ ($0 \leq t < k$) already found.

In addition, we need to ensure that $\vec{m}_k$ is linearly independent from all the row vectors that have already been determined so far, i.e., $\vec{m}_t$ ($0 \leq t < k$). Therefore, we perform a check whenever a solution $\vec{m}_k$ is returned. Let $M'$ be the matrix that consists of the row vectors determined already, and $M'_\perp$ be the sub-space orthogonal to $M'$, where $M'_\perp = I - M'^T(M'.M'^T)^{-1}.M'$. Then, $\vec{m}_k$ should satisfy the condition: $M'_\perp.\vec{m}_k \neq 0$.

The pseudo-code of our iteration-to-virtual core mapping algorithm is given in Algorithm 1.
**Algorithm 1** Determining mapping matrix $M$.  

**INPUT**: $l$-dimensional loop nest, separator vectors: $\Delta = (\vec{\delta}_1, \vec{\delta}_2, ..., \vec{\delta}_{s'} - 1)$, reuse matrix $R = (\vec{r}_0; \vec{r}_1; \vec{r}_2; ..., \vec{r}_{q-1})$  

**OUTPUT**: $M = (\vec{m}_0, \vec{m}_1, \vec{m}_3, ..., \vec{m}_{s'} - 1)$

1: /*Reuse vector grouping */
2: Expand $R$ as $R = (\vec{r}'_0; \vec{r}'_1; \vec{r}'_2; ...; \vec{r}'_{p-1})$;
3: Sort $R$ in a lexicographically ascending order based on their absolute values;
4: for each $\vec{\delta}_k$, where $1 \leq k \leq s' - 1$ do
5: Determine $g_k$;
6: end for
7: /*Determining $\vec{m}_0$ */
8: Find all the dependence vectors $D$
9: Select lexicographically smallest $l - 1$ linearly independent dependence vectors  
$D' = \{\vec{d}_1, \vec{d}_2, \cdots, \vec{d}_{l-1}\}$;
10: Set up equality constraints: $\vec{m}_0.\vec{d}_1 = 0; \vec{m}_0.\vec{d}_2 = 0; \cdots; \vec{m}_0.\vec{d}_{l-1} = 0$;
11: Set up legality constraints: $\forall \vec{d}' \in D - D' \vec{m}_0.\vec{d}' \geq 0$;
12: Set up locality constraints: $\forall \vec{r}' \in R - D' | \vec{m}_0.\vec{r}'| \leq q_0$;
13: Solving for minimal $q_0$ and lexicographically smallest $\vec{m}_0$
14: if no solution found then
15: Remove all the constraints related to the lexicographically largest reuse vector based on the absolute value;
16: Go to line 13
17: end if
18: /*Determining $\vec{m}_k$ ($1 \leq k \leq s$) */
19: for each group $g_k$ do
20: Select lexicographically smallest $l - 1$ linearly independent reuse vectors $R' = \{\vec{r}'_1, \vec{r}'_2, \cdots, \vec{r}'_{l-1}\}$ from $g_k$
21: Set up equality constraints: $\vec{m}_k.\vec{r}'_1 = 0; \vec{m}_k.\vec{r}'_2 = 0; \cdots; \vec{m}_k.\vec{r}'_{l-1} = 0$;
22: Set up legality constraints: $\forall \vec{d} \in D \vec{m}_0.\vec{d} \geq 0$;
23: Set up locality constraints: $\forall \vec{r}' \in R - R' \vec{m}_k.\vec{r}' \leq q_k$;
24: Set up linear dependency constraint for $\vec{m}_k$;
25: Solving for minimal $q_0$ and lexicographically smallest $\vec{m}_0$
26: if no solution found then
27: Remove all the constraints related to the lexicographically largest reuse vector based on the absolute value;
28: Go to line 25
29: end if
30: end for
31: return $M$
2.5.2 Virtual Core-to-Physical Core Mapping

Given an on-chip cache hierarchy of $s$ layers and a mapping matrix $M = (\vec{m}_0 \vec{m}_1 \cdots \vec{m}_{s'-1})^T$, where $s' = \min(l, s)$, our goal now is to determine a folding function $F$, denoted as $F = (f_0 \ f_1 \ \cdots \ f_{s'-1})^T$. As mentioned earlier, the main functionality of a folding function is to map each virtual core to a physical core (in the form of core vector). Recall that the virtual core $\vec{c}$ to which an iteration $\vec{i}$ gets mapped is given by $\vec{c} = M. \vec{i}$. Let $c_0$ denote the number of cores that are connected to an $L1$ cache in the target architecture and $c_i$ denote the number of caches at layer $i$ that are connected to the same cache at layer $i + 1$, where $1 \leq i \leq s' - 1$. The folding function $F$ can be calculated by computing each entry $f_j\ (0 \leq j \leq s' - 1)$ from right to left starting with $j = s' - 1$:

$$f_j = (\vec{m}_{s'-1}, \vec{i}/T_j) \mod c_j,$$

(2.10)

where $T_j$ is the block size in terms of the number of virtual cores. After applying $f_j$, every $T_j$ virtual cores are given the same cache ID at layer $j$ when $(1 \leq j \leq s' - 1)$, or the same core ID when $j = 0$. Note that, this mapping represents a block-wise distribution of virtual cores to physical cores. Once the folding function $F$ is determined, we map each iteration to a physical core. Figure 2.4(b) illustrates the iteration-to-physical core mapping for a two-level loop nest, for the sample multicore architecture shown in Figure 2.4(a). We want to make it clear that this block-wised distribution is only one method for selecting a folding function;

Figure 2.4: An illustration of iteration-to-physical core mapping for a two-level loop nest with a sample architecture.
Figure 2.5: Original code.

```c
for (i=1; i<=m; i++)
  for (j=0; j<=n; j++)
    b[i][j] = (a[i][j] + a[i+1][j+1] + a[i-1][j+2]);
```

Figure 2.6: Blocked code.

```c
for (ii=1; ii<=m; ii+s)
  for (jj=0; jj<=n; jj+t)
    for (i = ii; i <= min(ii+s, m); i++)
      for (j = jj; j <= min(jj+t, n); j++)
        b[i][j] = (a[i][j] + a[i+1][j+1] + a[i-1][j+2]);
```

Figure 2.7: Code generated by our scheme.

Figures 2.5, 2.6, 2.7

alternate methods are also possible. We also want to point out that this particular folding function preserves data locality (obtained in the virtual core space) in the physical core space.

2.5.3 Code Generation

In this section, we briefly discuss our code generation process, which consists of two steps. The first step is to perform a linear loop transformation for the iteration-to-virtual core mapping. The mapping matrix \( M \) obtained by applying Algorithm 1 is the key to achieve this transformation. Specifically, our linear loop transformation maps an iteration vector \( \mathbf{i} \) to a new iteration vector \( \mathbf{i}' \) by \( \mathbf{i}' = M \mathbf{i} \). The array

```c
for (ii=0; ii< ; ii+2*s)
  for (jj=0; jj<j+3*t){
    //running on core 0
    for (i = ii; i <= min(ii+s, m); i++)
      for (j = jj; j <= min(jj+t, n); j++)
        b[i][j] = a[i][j] + a[i+1][j+1] + a[i-1][j+2];
    //running on core 1
    for (i = ii; i <= min(ii+s, m); i++)
      for (j = jj+t; j <= (jj+2*t, n); j++)
        b[i][j] = a[i][j] + a[i+1][j+1] + a[i-1][j+2];
    .......
    //running on core 5
    for (i = ii+s; i <= min(ii+2*s, m); i++)
      for (j = jj+2*t; j <= min(jj+3*t, n); j++)
        b[i][j] = a[i][j] + a[i+1][j+1] + a[i-1][j+2];
  }
```

Figure 2.7: Code generated by our scheme.
Our second step is to perform loop blocking based on the transformed loop nest and the folding function (obtained in Section 2.5.2). Different from any existing work on loop blocking, we ”unroll” the blocks and set the thread-to-core affinity to each unrolled block in order to enforce the virtual core-to-physical core mapping. An example of the generated code is illustrated in Figure 2.7.

2.6 Scheduling for Dependence-Free Loops

In case that the target loop nest is dependence-free, we propose a novel loop scheduling scheme to maximize the performance of the shared caches in the system. This can be achieved by reducing the reuse distance between two iterations that

```c
for (ii=0; ii< i+2*s)
for (jj=0; jj< j+3*t)
    //running on core 0
    for (i = ii; i <= min(i+s, m); i++)
        for (j = jj; j <= min(j+t, n); j++)
            b[i][j] = a[i][j] + a[i+1][j+1] + a[i-1][j+2];
    //running on core 1
    for (i = min(ii+s, m); i >= ii; i--)
        for (j = (jj+2*t, n); j >= jj+t, j++)
            b[i][j] = a[i][j] + a[i+1][j+1] + a[i-1][j+2];
    //running on core 5
    for (i = ii+s; i <= min(i+2*s, m); i++)
        for (j = jj+2*t; j <= min(jj+3*t, n); j++)
            b[i][j] = a[i][j] + a[i+1][j+1] + a[i-1][j+2];

Figure 2.8: Code generated by our scheduling scheme.

Figure 2.9: Illustration of our scheduling scheme (a) after the iteration-to-physical core mapping (b).

reference \( \vec{r} \), where \( \vec{r}' = A\vec{r} + \vec{a} \), is changed accordingly to \( \vec{r}' \), i.e., \( \vec{r}' = AM. \vec{i} + \vec{a} \).
are mapped to two different cores that share an on-chip cache. In other words, our goal is to come up with a schedule for each core such that, if two cores share a data block, they share it within a small period of time, thereby increasing chances of catching the reused data block in the shared cache. A distinguishing characteristic of our proposed approach compared to previous loop transformation and scheduling theory, is that we determine a \textit{customized schedule} for each core yet these schedules are adjusted with respect to each other to reduce inter-core data reuse distances (Figure 2.9). In mathematical terms, assuming that there are \( n \) cores in the target architecture, we determine \( n \) row vectors called \textit{scheduling vectors}, \( \vec{S}_0 \) through \( \vec{S}_{n-1} \), such that all data dependences are satisfied and inter-core reuse distances are minimized. A scheduling vector takes an iteration as input and determines a single value (called \textit{schedule slot}) that determines the point in time at which that iteration is to be executed. For example, if \( \vec{i} \) is a loop iteration and \( \vec{S} \) is the scheduling vector, \( \vec{S}.\vec{i} = \tau \) represents the \textit{schedule slot} for \( \vec{i} \). The scheduling vector \( \vec{S} \) (for each core) applied to the iterations mapped to that core, exploits data reuse at runtime across the cores that share on-chip caches. Our schedules should satisfy the following constraint:

\textbf{Locality Constraint.} If two iterations \( \vec{i}_1 \) and \( \vec{i}_2 \) \((\vec{i}_1 \prec \vec{i}_2)\) mapped to different cores have data reuse between them, the locality constraint implies that the difference between their schedule slots should be minimum: \( \vec{S}_2.\vec{i}_2 - \vec{S}_1.\vec{i}_1 < q \), where \( q \) is subjected to be minimized. This constraint can be referred to as the \textit{inter-core locality constraint}. If \( \vec{i}_1 \) and \( \vec{i}_2 \) belong to the same core however (i.e., \( \vec{S}_1 = \vec{S}_2 \)), we refer to the same constraint as \textit{intra-core locality constraint}. In our approach, the scheduling vectors as well as the minimum value of \( q \) that satisfies the locality constraint are determined using the Farkas Lemma\(^6\) and the integer FME. Figure 2.8 illustrates the generated code in our scheduling scheme.

\(^6\)Farkas Lemma [24] is a result in mathematics stating that a vector is either in a given cone or that there exists a hyperplane separating the vector from the cone, but not both. Let \( D \) be a non-empty polyhedron defined by \( p \) affine inequalities, \( a_k.x + b_k \geq 0 \), \( 1 \leq k \leq p \). An affine form \( \psi \) is non-negative everywhere in \( D \) iff it is a positive affine combination of the faces: \( \psi(x) = \lambda_0 + \Sigma \lambda_k (a_k.x + b_k) \), \( \lambda_k \geq 0 \). The non-negative constants \( \lambda_k \) are referred to as the Farkas Multipliers. Since the loop bounds can be written in the form of \( A.\vec{i} + B \geq 0 \), where \( A \) and \( B \) are constant matrices, by using the Farkas Lemma and the integer FME, one can transform the original linear system to another system where loop iterator \( \vec{i} \) is eliminated.
2.7 Experimental Evaluation

1. Setup and Applications

We performed all our experiments on an Intel Dunnington multicore machine. The relevant details of this architecture are presented in Table 2.1. In our experiments, all 12 cores are used. Dunnington has a cache hierarchy similar to the three-layer architecture shown in Figure 1.1(c), except that it has 12 L1s, 6 L2s, and 2 L3s. Each L3 is connected to 3 L2s, and each L2 is connected to 2 L1s. The applications used in our study are listed in Table 2.2. It shows cache statistics and parallel execution times under the default (original) version (explained below). The first six applications represent kernels that are frequently used in data-intensive/scientific computing; and the last two codes are taken from a computer game development toolkit. The total sizes of the datasets manipulated by these applications varied between 24.4MB and 58.2MB. Most of the results presented in the following subsections are percentage improvements (in cache misses or execution cycles) over the default version. Most of our experiments are performed using four different versions explained below. In our parallelization/mapping scheme, we used the separator vector selection strategy explained in Section 2.5.1.

- **Default.** In this version, iterations of a parallel loop nest are distributed in a block fashion across cores and each core executes the iterations assigned to it in their original execution sequence. In a sense, this version represents a straightforward mapping strategy combined with the original order of iterations mapped to each core.

- **Default+.** This is similar to the default version except that, after mapping, computations mapped to each core are optimized using loop permutation and loop tiling. More explanations on this version will be given later in this section.

- **Mapping.** This is our proposed parallelization/mapping strategy. In our experiments, we evaluated three variants of this version:

  - **L3.** This version aims at optimizing only for the L3 cache.

  - **L2+L3.** This is a version in which our approach targets only the L2 and L3 layers.

  - **L1+L2+L3.** This represents a version where our approach targets all three
Table 2.1: Important characteristics of the Intel Dunnington architecture.

<table>
<thead>
<tr>
<th>12 cores (2 sockets)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32KB, 8-way, 64-byte line size, 3 cycle latency</td>
</tr>
<tr>
<td>3MB, 12-way, 64-byte line size, 12 cycle latency</td>
</tr>
<tr>
<td>12MB, 16-way, 64-byte line size, 40 cycle latency</td>
</tr>
<tr>
<td>about 85 ns</td>
</tr>
</tbody>
</table>

Figure 2.10: Improvements in cache misses and execution cycles.

layers in the target on-chip cache hierarchy.

Note that, while this version optimizes mapping, for the iterations mapped to each core, it maintains their original execution order (as in the case of the default strategy).

- **Mapping+Scheduling.** This version first uses our parallelization/mapping scheme and then our scheduling strategy, explained in Section 2.6.

We used the SUIF compiler infrastructure [27] (as a source-to-source translator) to implement these versions. All these versions have been compiled with the native Intel compiler with -O3. Consequently, even the default version incorporates low level (single core centric) code optimizations. The different versions also execute the same set of iterations in parallel. Therefore, performance differences among different versions are only due to different mapping and scheduling strategies they adopt.

2. Results

Our first set of results are presented in Figure 2.10 and plot the percentage L1 miss, L2 miss, L3 miss and execution time improvements brought by versions
Figure 2.11: Comparison against conventional optimization and the impact of scheduling.

Figure 2.12: Results compared to hierarchical tiling and polyhedral model.

Figure 2.13: Sensitivity to the selection of separator vectors.
<table>
<thead>
<tr>
<th>Appl.</th>
<th>Brief Description</th>
<th>Misses (%)</th>
<th>Exec Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>L1</td>
<td>L2</td>
</tr>
<tr>
<td>mwxm</td>
<td>Matrix-matrix multiplication</td>
<td>37.1</td>
<td>16.8</td>
</tr>
<tr>
<td>rbo</td>
<td>Red-black overrelaxation</td>
<td>27.7</td>
<td>19.2</td>
</tr>
<tr>
<td>lu</td>
<td>LU decomposition</td>
<td>18.9</td>
<td>26.3</td>
</tr>
<tr>
<td>mgrid</td>
<td>Multigrid solver (SPECOMP [28])</td>
<td>16.4</td>
<td>20.8</td>
</tr>
<tr>
<td>cg</td>
<td>Conjugate gradient (NAS [29])</td>
<td>8.8</td>
<td>13.2</td>
</tr>
<tr>
<td>fma3d</td>
<td>Finite element crash simulation (SPECOMP [28])</td>
<td>9.5</td>
<td>9.6</td>
</tr>
<tr>
<td>rcomm</td>
<td>Rendering through predefined objects like sprites</td>
<td>37.1</td>
<td>6.6</td>
</tr>
<tr>
<td>draw 1.1</td>
<td>Vector drawing library</td>
<td>26.8</td>
<td>19.2</td>
</tr>
</tbody>
</table>

Table 2.2: Benchmarks.

L3, L2+L3, and L1+L2+L3 over the default version. There are two important observations one can make from these results. First, all optimized versions generate better results than the default version. Second, and more importantly, we can see that optimizing for the entire on-chip cache hierarchy is very important. Specifically, average execution time improvements brought by the L3, L2+L3, and L1+L2+L3 versions over the default version are, respectively, 5.4%, 9.3%, and 14.1%. This means, for example, that optimizing for L3 alone does not guarantee very good performance at the L2 and L1 layers.

Figure 2.11 plots the performance of three different versions. The first group of bars (named Default+) represents a version that uses block distribution (mapping) of iterations over cores (as in the case of the default version) but employs a set of well-known cache locality optimizations designed to improve L1 performance. These optimizations are applied to each core in isolation and include loop permutation and iteration space tiling. The former changes the order of loops in a nest so that data reuses are brought into inner loop positions to the extent allowed by data dependences. The latter on the other hand generates a blocked version of the code. Specifically, it employs the data-centric tiling strategy proposed by Kodukula et al [30], which (based on our preliminary experiments) generated better results in general than traditional control-centric tiling schemes. The second group of bars (named Mapping) is the L1+L2+L3 variant explained earlier (reproduced here for comparison). Finally, the last group of bars show the results obtained when using the Mapping+Scheduling version (i.e., the L1+L2+L3 version followed by our scheduling scheme). Our first observation is that using conventional data locality optimizations (instead of the hierarchy and sharing aware mapping and scheduling) generates about 7.3% improvement in execution times on average. This is
much lower compared to the average execution time improvement of 14.1% produced by the Mapping version. In other words, careful mapping of iterations to cores is very important in multicores. The second observation is that the Mapping+Scheduling version achieves an average execution time improvement of about 18%, which means careful scheduling of loop iterations (assigned to cores) considering timing of data sharing can bring further benefits over mapping alone.

3. Comparison against Hierarchical Tiling and Polyhedral Approach

In our next set of experiments, we compare our approach against two alternate schemes. The first of these is a hierarchical tiling approach where an iteration space is tiled multiple times (hierarchically) considering the underlying cache topology. For example, when targeting the Dunnington architecture, the loop nest is tiled three times; the first tiling is based on L3 cache capacity; the second one is based on L2 capacity; and the last one is based on L1 capacity. After this tiling, the outer tiles (i.e., those based on L1) are distributed across cores. The reason why we perform experiments with this version is to see whether a simple hierarchical tiling strategy generates similar results to our mapping scheme. The second scheme that we test is based on a recent study [31] which implements a polyhedral approach for code mapping in multicores. From the results in Figure 2.12, one can observe that our scheme (Mapping+Scheduling) generates better results than hierarchical tiling for all application programs tested. This is because it takes into account not just cache capacities (as in the case of hierarchical tiling) but also cache topology, which in turn leads to better exploitation of data reuse across cores. As compared to the polyhedral approach, our scheme generates better results in six of our eight applications, resulting in an additional execution time improvement of 7% on average in these applications. In four of the applications we generate better results, the code transformations determined by the two approaches were different. In the remaining two cases we perform better, the main reason was the high overheads involved with the code generated using the polyhedral model. Compared to our linear algebraic model, a polyhedral model has two main drawbacks. First, it generally has higher overheads at runtime due to the large number of checks, branch operations, and complex loop bound computations. Second, transformations applied to a statement necessarily touch all instances of that statement, leading to both code bloat and increased loop overhead. For example, when the approach in [31] is applied
to the codes in our experimental suite, we observed about a 3x increase in code size, which may not be acceptable in certain environments. The polyhedral model generated slightly better results (around 1%) in only two benchmarks (mgrid and draw 1.1), due to the fact that it produced a different transformation than ours. Overall, the results plotted in Figures 2.10, 2.11, and 2.12 clearly underline the importance of cache hierarchy aware data locality optimization for multicores.

4. Impact of Separator Vectors

Figure 2.13 plots the improvements in execution times (over the default version) with different selection vectors. On the x-axis of this graph, (0, 0) corresponds to the separator vectors determined by the strategy in Section 2.5.1. (−, 0) corresponds to moving the first separator vector toward left (i.e., reducing the number of reuses to be exploited in L1 layer and increasing the number of reuses to be exploited in L2 layer), whereas (+, 0) corresponds to moving the first separator vector toward right (i.e., increasing the number of reuses to be exploited in L1 and reducing the number of reuses to be exploited in L2 layer). In both the cases, the second separator vector is kept the same. Similarly, (0, −) and (0, +) correspond to moving the second separator vector toward left and right, respectively. One can see from these results that both (−, 0) and (+, 0) generate lower performance than (0, 0), though for different reasons. (0, 0) utilizes the available L1 space very well and trying to increase the number of reuses to be exploited in L1 layer (i.e., (+, 0)) puts more pressure in the cache and, as a result, L1 caches overflow, resulting in performance degradation. On the other hand, (−, 0) leads to underutilization of the L1 cache space. The same observation also explains why we have relatively lower performance when (0, +) or (0, −) is adopted.
Chapter 3

Optimizing Data Layouts for Parallel Computation on Multicores

3.1 Overview of Our Data Layout Transformation

In this section, we give a high-level view of our data layout transformation strategy, which targets localized arrays in a parallel loop nest. Generally speaking, a localized array has two important characteristics. First, the data accessed by each thread from this array are local, i.e., each thread operates mostly on its own portion of data and there is little data sharing (if at all) across different threads regarding this array. Second, if the data portion accessed by a thread is further divided into sub-blocks, these sub-blocks will be locally accessed by different computation sets (a computation set is a group of iterations) within this thread at different times, i.e., each computation set operates mostly on its own portion of data and there is little data sharing across different computation sets.

These two characteristics imply that (i) the reuse distance (the number of iterations between two consecutive uses of a data) in a localized array is small, and (ii) the data are localized within each thread (space-wise) as well as within each computation set to be executed by a thread (time-wise). Figure 3.1 illustrates an example data access pattern exhibited by 8 threads on a localized array, where $t_{ij}$ denotes a data block that is local to the computation set $j$ of thread $i$. Each data
block is assumed to contain two sub-rows, as shown in the figure. One can employ a conventional layout such as row-major to store this array in the linear memory (i.e., to map data space to memory space), as depicted by Figure 3.2(a). The drawback of this strategy is that, it makes the elements even from one data block (locally accessed by a computation set) scattered all over the memory space, which can cause conflict misses even among data accesses within a thread. Figure 3.2(b) shows an alternate layout strategy where the data local to a thread are stored consecutively. While this can reduce misses caused by the data accesses from each thread, one can do better if accesses coming from different threads are considered “together”. Consider now the layout in Figure 3.2(c), which assumes that a single shared on-chip cache is used by all threads and its capacity is equal to the size of eight sub-rows. In this layout, the data elements expected to be accessed by different threads in a given period of time are stored consecutively based on the cache size, which minimizes the total memory footprint generated by all threads and thus reducing both conflict and capacity misses in the shared cache.

However, multicores in the market today usually have multiple shared caches with a variety of on-chip cache hierarchies and capacities. To minimize the conflict and capacity misses in all shared caches, the target cache hierarchy (as well as capacity at each layer) needs to be taken into account. Figure 3.2(d) illustrates such an example layout (in an abstract form) for the cache topology shown in Figure 1.1(c) (assuming that L2 is two times larger than L1 and L3 is two times larger than L2, for illustrative purposes), where each box indicates the data locally accessed by a thread at a given period of time. Note that, this layout is hierarchical and can help reduce the TLB misses and improve the effectiveness of hardware prefetching employed by the underlying architecture as well (in addition to improving cache performance).

To achieve such customized (architecture and inter-thread access specific) memory layouts, we have designed a novel data layout transformation, which consists of the following four steps for each array in a given parallel loop nest:

- **Localization test**: tests whether the array is localized; and if so, perform the following steps for it;
- **Tiling determination**: decides a suitable tiling, in which the array is partitioned into disjoint portions such that each portion is accessed mostly by one
Figure 3.1: A sample access pattern exhibited by 8 threads on a two-dimensional array.

thread, and each sub-block in a portion is processed mostly by one computation set of a thread.

- **Transformation determination**: determines a transformation function, which converts the default (original) memory layout to the new layout found in the previous step; and

- **Memory mapping**: maps the tiles to the memory space considering cache hierarchy.

In case that there are multiple loop nests or if the target loop nest is imperfectly nested, for each array, we first identify the loop nests in which this array is localized, and then choose a *reference loop nest* among these candidates for layout optimization purposes, which has the most "localized access pattern" for this array considering the "weight" of loop nest, e.g., the loop count and the number of array references optimized. However, if an array is involved in an external function call such as `memcpy()`, our optimization will not be applied even if it is localized, since we cannot change its data layout completely. Note that, our inter-thread data layout optimization takes place after loop transformation and parallelization, and it is orthogonal to other code optimizations such as vectorization. We apply our technique to a single array at a time, and optimization on multiple arrays at the same time is on our future research agenda.
3.2 Polyhedral Representation

Since our focus is on affine loop nests, we start our discussion by presenting the polyhedral model [32] employed by our data layout transformation strategy.

3.2.1 Overview of Hyperplanes

Hyperplane is an important concept in any polyhedral model. In an $m$-dimensional space represented by a vector of variables $\vec{b} = (b_1, b_2, \ldots, b_m)^T$, an affine hyperplane $g$ is defined as: $g_1b_1 + g_2b_2 + \cdots + g_mb_m = c$, where $g_1, g_2, \ldots, g_m$ are rational numbers called hyperplane coefficients and $c$ is a rational number called hyperplane constant [33]. The hyperplane vector $\vec{g} = (g_1, g_2, \ldots, g_m)$ is normal to the hyperplane and defines a hyperplane family where “member” hyperplanes have the same hyperplane vector but different values of $c$. While an affine equality represents an $(m-1)$-dimensional affine subspace, an affine inequality such as $\vec{g} \cdot \vec{b} \geq c$ represents a halfspace including points that lie on one side of a hyperplane in an $m$-dimensional space. The intersection of finitely many half-spaces forms a polyhedron, and a bounded polyhedron is called a polytope, denoted as $M \cdot \vec{b} \geq \vec{c}$, where $\vec{c}$ is a vector of constants, and $M$ is a matrix in which every row defines a hyperplane.
that bounds the polytope.

### 3.2.2 Polyhedral Model

In the polyhedral model, an \( n \)-level loop nest represents an \( n \)-dimensional iteration space \( I \), which can be denoted using an iteration vector \( \vec{i} = (i_1, i_2, \ldots, i_n)^T \), where \( i_k \) is the iterator of the \( k \)th loop (starting from the outermost one). The condition \( L_k \leq i_k \leq U_k \) holds for each \( i_k \), where \( L_k \) and \( U_k \) are the corresponding lower and upper loop bounds, respectively. For a statement \( s \) within such a loop nest, the set of values of the iterations for which \( s \) has to be executed can always be specified by a set of affine linear inequalities derived from loop indices (to simplify the notation, we omit the loop-invariant parameters in the rest of our discussion). These inequalities form an iteration space polytope that can be expressed in the matrix form as: \( I_s \cdot \vec{i} \geq \vec{c}_s \), where each row in matrix \( I_s \) and vector \( \vec{c}_s \) together defines a hyperplane, and each point in the polytope corresponds to a dynamic instance of statement \( s \) in the program execution.

Similarly, an \( m \)-dimensional array \( A \) represents an \( m \)-dimensional data space \( D_A \), which can be denoted using a data (index) vector \( \vec{a} = (a_1, a_2, \ldots, a_m)^T \). Each element of this vector corresponds to a subscript (index) expression in an access to array \( A \), e.g., \( A[a_1, a_2, \ldots, a_m] \). If the \( r \)th reference to array \( A \) (denoted using \( A_r \)) is affine, \( \vec{a} \) can be expressed in the matrix form as: \( \vec{a} = Q_{Ar} \cdot \vec{i} + q_{Ar} \), where \( Q_{Ar} \) is the access matrix and \( q_{Ar} \) is the offset vector. The access matrix \( Q_{Ar} \) stands for a mapping from the iteration space to the data space of array \( A \) through its \( r \)th reference. For example, given the following code:

```plaintext
for(i1 = 0; i1 ≤ 100; i1 + +)
  for(i2 = 0; i2 ≤ 100; i2 + +)
    S1 : A[i1 + 3, i2 - 1] = A[i1, i2] + B[i1, i2];
...
```

the iteration space polytope of statement \( S1 \) defined by \( \{i_1, i_2 \mid 0 \leq i_1 \leq 100 \land 0 \leq \)
\( i_2 \leq 100 \) can be written in the matrix form as:

\[
\begin{bmatrix}
1 & 0 \\
-1 & 0 \\
0 & 1 \\
0 & -1
\end{bmatrix} \cdot \vec{i} \geq \begin{bmatrix}
0 \\
-100 \\
0 \\
-100
\end{bmatrix},
\]

and the first reference \( A[i_1 + 3, i_2 - 1] \) of array \( A \) in the matrix representation can be expressed as:

\[
A_1 : \quad \vec{a} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \vec{i} + \begin{bmatrix} 3 \\ -1 \end{bmatrix},
\]

where \( \vec{i} = (i_1, i_2)^T \) is the iteration vector, and \( \vec{a} = (a_1, a_2)^T \) is the data vector with \( a_1 = i_1 + 3 \) and \( a_2 = i_2 - 1 \).

### 3.3 Hyperplane Transformation

Hyperplane transformation translates a hyperplane in the iteration space (an iteration hyperplane) into a hyperplane in the data space (a data hyperplane) of an array through its reference(s). The reason why we discuss this transformation is that it helps us find a set of data hyperplanes for a localized array in a loop nest whose iteration space is partitioned by a group of iteration hyperplanes; these data hyperplanes can then be used to form a tiled version of the array and track the portions/sub-blocks touched by different threads as well as different computation sets (within a given thread). In other words, hyperplane transformation represents a computation-to-data mapping for an array.

Given an iteration hyperplane \( h : \vec{h} \cdot \vec{i} = c \), where \( \vec{h} \) is the hyperplane vector, and an array reference \( A_r : \vec{a} = Q_{Ar} \cdot \vec{i} + q_{Ar} \), if the access matrix \( Q_{Ar} \) is nonsingular, \( \vec{i} \) can be expressed as \( \vec{i} = Q_{Ar}^{-1} \cdot (\vec{a} - q_{Ar}) \). By eliminating \( \vec{i} \) in \( \vec{h} \cdot \vec{i} = c \), we can obtain a data hyperplane \( h' \) as:

\[
h' : \quad \vec{h}' \cdot \vec{a} = c',
\]

where \( \vec{h}' = \vec{h} \cdot Q_{Ar}^{-1} \) and \( c' = c + \vec{h} \cdot Q_{Ar}^{-1} \cdot q_{Ar} \). In the case that \( Q_{Ar} \) is singular but fully ranked, i.e., the number of dimensions of array \( A \) is unequal to the number of dimensions of the iteration space \( I \), \( \text{dim}(A) \neq \text{dim}(I) \), one can still find a
corresponding data hyperplane $h'$ by using the following strategy:

**Case 1:** $\dim(A) > \dim(I)$. Assuming that $\dim(A) = m$ and $\dim(I) = n$, we decompose the access matrix $Q_{Ar}$ into two parts, $Q_{1Ar}$ and $Q_{2Ar}$, where $Q_{1Ar}$ consists of $n$ independent rows chosen from $Q_{Ar}$, and $Q_{2Ar}$ contains the rest $(m-n)$ rows. By selecting variables from the data vector $\vec{a}$ and constants from the offset vector $\vec{q}_{Ar}$ that correspond to the rows in the matrix $Q_{1Ar}$, we can create a new reference $A_{r}'$: $\vec{a}_1 = Q_{1Ar} \cdot \vec{i} + \vec{q}_{1Ar}$, where $\vec{a}_1$ and $\vec{q}_{1Ar}$ are the new data and offset vectors, respectively. Since the matrix $Q_{1Ar}$ is nonsingular, we then use $\vec{a}_1$, $Q_{1Ar}^{-1}$ and $q_{1Ar}$ to obtain a data hyperplane $h'$, which is essentially a hyperplane obtained in the sub $n$-dimensional data space of the original array. For example, suppose that array reference $A[i_1, i_1 + i_2, i_2]$ resides in a two-level loop nest where $\vec{i} = (i_1, i_2)^T$. The matrix representation of this reference is:

$$A_1 : \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}^T = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \cdot (i_1, i_2)^T + \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$  

To find the data hyperplane for the iteration hyperplane $i_1 = 4$, we first create the reference:

$$A_{r}'_1 : \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}^T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot (i_1, i_2)^T + \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$  

and then use Eq. (3.1) to find a data hyperplane through the new reference, which is $a_1 = 4$.

**Case 2:** $\dim(A) < \dim(I)$. In this case, we construct a new array reference $A_{r}'$ from $A_r$, by adding $(n-m)$ rows to the access matrix $Q_{Ar}$, which are linearly independent of the $m$ rows already in $Q_{Ar}$, and adding $(n-m)$ variables to $\vec{a}$ as well as $(n-m)$ constants of value 0 to $q_{Ar}$. This newly-created reference $A_{r}'$ can be represented as: $\vec{a}' = Q_{Ar}' \cdot \vec{i} + q_{Ar}'$, where $\vec{a}'$, $Q_{Ar}'$, $q_{Ar}'$ correspond to the new data vector, access matrix, and offset vector, respectively. Since $Q_{Ar}'$ is nonsingular, we can then use $\vec{a}'$, $Q_{Ar}'^{-1}$ and $q_{Ar}'$ to obtain a data hyperplane $h'$, which is essentially a hyperplane obtained in the super $n$-dimensional data space of the original array. $h'$ is valid for the original data space if and only if the coefficients of the added $(n-m)$ dimensions are 0. Otherwise, $h'$ needs to be dropped, and there is no data
hyperplane solution for the iteration hyperplane \( h \) through \( A_r \). As an example, suppose that array references \( A[i_1, i_2] \) and \( A[i_2 + 2, i_3 - 1] \) reside in a three-level loop nest \( \vec{i} = (i_1, i_2, i_3)^T \). The matrix representations of these two references are:

\[
A_1 : \quad (a_1, a_2)^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \cdot (i_1, i_2, i_3)^T + \begin{bmatrix} 0 \\ 0 \end{bmatrix},
\]

\[
A_2 : \quad (a_1, a_2)^T = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot (i_1, i_2, i_3)^T + \begin{bmatrix} 2 \\ -1 \end{bmatrix}.
\]

To find the data hyperplane for the iteration hyperplane \( i_1 = 4 \), we first transform the two references to the following forms:

\[
A'_1 : \quad (a_1, a_2, a_3)^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot (i_1, i_2, i_3)^T + \begin{bmatrix} 0 \\ 0 \end{bmatrix},
\]

\[
A'_2 : \quad (a_1, a_2, a_3)^T = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \cdot (i_1, i_2, i_3)^T + \begin{bmatrix} 2 \\ -1 \\ 0 \end{bmatrix},
\]

and then use Eq. (3.1) to obtain data hyperplanes, which are \( a_1 = 4 \) and \( a_3 = 4 \), respectively. Note that, only \( a_1 = 4 \) is valid for the original data space of array \( A \), since the coefficient of the added (third) dimension of hyperplane \( a_3 = 4 \) is not 0.

In the case that no data hyperplane exists for any given iteration hyperplane through reference \( A_r \), i.e., its access matrix is not invertible or the obtained data hyperplane is not valid, we simply exclude \( A_r \) when determining a tiling for array \( A \), since this indicates that different threads or computation sets share a large amount of data in array \( A \) through \( A_r \), such as \( A[i_2 + 2, i_3 - 1] \) in the above example. An array is said to be \textit{localized}, if its tiling is decided by considering all its references; otherwise, it is termed as \textit{partially localized}.
3.4 Inter-thread Layout Optimization

This section gives the technical details of the most important contribution of this work: a novel data layout strategy for multithreaded programs that execute on multicore with on-chip cache hierarchies.

3.4.1 Localization Test

For a loop nest, its iteration hyperplanes can be classified into two types: iteration-space hyperplane and iteration-time hyperplane. While the former participates in partitioning of the iteration space into parallel regions assigned to different threads, the latter takes part in dividing each parallel region further into computation sets to be executed sequentially by a single thread. The loops involved in the parallelization determined by iteration-space hyperplanes are usually at outermost positions in a loop nest, whereas the loops involved in the sequential execution determined by iteration-time hyperplanes are usually at innermost positions. According to our definition of localized array, we conduct localization tests using both iteration-space hyperplanes and iteration-time hyperplanes (which are obtained from the loop parallelization step applied before our optimization is invoked).

Specifically, for each reference to an array, we test the reuse distance between iterations across different threads (space-wise) as well as the reuse distance between iterations across different computation sets within each thread (time-wise). As the reuse distance can be represented by an integer vector called reuse vector [34] [35] in the iteration space, given a hyperplane, we calculate the distance this reuse vector traverses along the normal of the hyperplane, and test whether the obtained value is less than a preset threshold. The reason for doing so is that, this value represents the width of the band across the boundary of two iteration regions separated by this hyperplane (see Figure 3.3), which indicates their communication volume or the amount of data shared. If this bandwidth is small, i.e., the amount of common data accessed by two regions is small, then the data are said to be localized within each region.

Let $H_s$ and $H_t$ be the set of iteration-space and iteration-time hyperplanes for a loop nest, respectively, and $R_A$ be the set of reuse vectors (containing distance
Figure 3.3: Band formed around a hyperplane (solid line) in the iteration space by reuse vectors (dashed lines) across the hyperplane.

Figure 3.4: Illustration of the distances (dashed lines) between array references (filled circles) and a hyperplane (solid line) in the data space.

values) for array $A$.\footnote{In the case that the reuse distance value along a particular dimension is not constant, we conservatively use the corresponding loop bound of that dimension as the distance value.} The space-wise localization test can be expressed as:

$$\forall h \in H_s, \quad \forall \vec{r} \in R_A, \quad \vec{h} \cdot \vec{r}^T \leq q_1,$$

and the time-wise localization test can be expressed as:

$$\exists h \in H_t, \quad \forall \vec{r} \in R_A, \quad \vec{h} \cdot \vec{r}^T \leq q_2,$$

where $\vec{h}$ is the hyperplane vector of $h$. Only when both these conditions captured by Eq. (3.2) and Eq. (3.3) hold true, we say that array $A$ is localized. Note that, if there exists one iteration-time hyperplane that satisfies the condition, we consider the array as time-wise localized (see Eq. (3.3)).

In our current implementation, $q_1$ (for space-wise test) is set to an experimentally-determined value; however, $q_2$ (for time-wise test) is chosen according to the last level cache capacity. Specifically, assuming that the last level cache is shared by $T$ threads and its capacity is $C$, then $q_2$ is calculated based on the following equation: $q_2 \times lh \times x = C/T$, where $lh$ represents the estimated length/area of the hyperplane being tested, using loop bound information, and $x$ denotes the size of data accessed by one loop iteration. The equation means that, ideally, we want the data shared by two successive computation blocks (represented by the band formed around the iteration-time hyperplane) to be captured in the cache.
3.4.2 Tiling Determination

We determine a tiling for a localized array using the hyperplane transformation explained in Section 3.3. A data hyperplane is called *data-space hyperplane* if it is transformed from an iteration-space hyperplane (through some reference), and *data-time hyperplane* if it is obtained from an iteration-time hyperplane. The tiling of this array, which matches the localized data access pattern, is formed by both data-space and data-time hyperplanes. Specifically, for an $m$-dimensional array in an $n$-level loop nest, let $(h_1, h_2, \cdots, h_k)$ be $k$ iteration-space hyperplanes that decide the parallelization, and $(h_{k+1}, h_{k+2}, \cdots, h_n)$ be $(n-k)$ iteration-time hyperplanes that decide the sequential execution of each parallel region. To obtain data-space hyperplanes, if $k \leq (m - 1)$, we directly use $k$ iteration-space hyperplanes; otherwise, we choose $(m - 1)$ iteration-space hyperplanes along which the communication volume is the least. To obtain data-time hyperplanes, if $k \leq (m - 1)$, we choose the iteration-time hyperplanes that correspond to the innermost $(m - k)$ loops; otherwise, we use $h_n$ directly. In the case that $m > n$, we can always find $(m - n)$ hyperplanes that are linearly independent from each other, and from the hyperplanes determined (similar to the strategy mentioned in Case 2 in Section 3.3), to form an $m$-dimensional tiling, and we classify these additional $(m - n)$ hyperplanes as data-time hyperplanes.

Note that, if a localized array has one reference in the loop nest, only one data hyperplane (space or time) will be obtained from the corresponding iteration hyperplane. However, if there are more than one reference, multiple data hyperplanes may exist, where we need to select one that results in the best localized partition in the data space, i.e., the largest distance between the data elements separated by this hyperplane, which are accessed in the same iteration, is the smallest. Suppose that localized array $A$ has $w$ references and $h$ is an iteration hyperplane. For each data hyperplane $h'_r$ transformed through reference $r$ ($r \in w$), we first calculate its maximum distance $d_r$ as follows:

$$ d_r = \max\{|\vec{h}'_r \cdot (\vec{a}_i - \vec{a}_r)^T|\}, \quad \forall i \in w, $$

(3.4)

where $(\vec{a}_i - \vec{a}_r)^T$ is the *data distance vector* between references $A_i$ and $A_r$, and then choose the data hyperplane whose $d_r$ is the smallest.
denotes the distance between reference $A_i$ and $h'_r$. Clearly, the distance between $A_r$ and $h'_r$ is 0, since $A_r$ is right on the hyperplane $h'_r$. Figure 3.4 illustrates the distances between a data hyperplane and four references to array $A$ in the iteration $(i,j)$ of the following loop nest.

\[
\begin{align*}
\text{for } (i = 0; \ i < 16; \ i++) \\
& \quad \quad \text{for } (j = 0; \ j < 16; \ j++) \\
& \quad \quad \quad B[i][j] = (A[i][j] + A[j][i] + A[i+1][j] \\
& \quad \quad \quad + A[i][j+1])/4.0;
\end{align*}
\]

### 3.4.3 Transformation Function Determination

Once the tiling is found for a localized array, i.e., the data-space and data-time hyperplanes are determined, we formulate a transformation function that translates the original array index space into the tiled array index space. Assuming that $\{h'_1, h'_2, \ldots, h'_m\}$ are the $m$ tiling hyperplanes found for an $m$-dimensional array $A$, the transformation matrix $H$ can be obtained using the method presented in [36], and the transformation function can be expressed as$^2$:

\[
f(\vec{a}) = \begin{pmatrix}
[H \cdot \vec{a}] \\
\vec{a} - H^{-1} \cdot [H \cdot \vec{a}]
\end{pmatrix},
\]

where $\vec{a}$ contains coordinates in the original data space. In this expression, $[H \cdot \vec{a}]$ represents the tile coordinates in the transformed data space, and $\vec{a} - H^{-1} \cdot [H \cdot \vec{a}]$ denotes the tile offsets. In other words, the $m$-dimensional data space is transformed to a new $2m$-dimensional data space.

To illustrate how we find a tiling and the corresponding transformation function, let us consider array $A$ in the loop nest shown in Section 3.4.2 again, which has four references: $\vec{a}_1 = (i, j), \vec{a}_2 = (j, i)^T, \vec{a}_3 = (i+1, j)^T$ and $\vec{a}_4 = (i, j+1)^T$. Their access matrices are:

\[
Q_{\vec{a}_1} = Q_{\vec{a}_3} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \text{and} \quad Q_{\vec{a}_2} = Q_{\vec{a}_4} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.
\]

$^2$[\ ] operator in this case is applied to each element of a vector.
Suppose that we have iteration-space hyperplane \((1, 1)\) and iteration-time hyperplane \((0, 1)\), and our thresholds \(q_1\) and \(q_2\) are set to 3 and 20, respectively. We first test all the reuse vectors \(R_A = \{(i - j, j - i)^T, (1, 0)^T, (0, 1)^T, (j - i - 1, i - j)^T, (j - i, i - j - 1)^T, (1, -1)^T\}\) with respect to the iteration-space and iteration-time hyperplanes \(\{(1, 1), (0, 1)\}\) using Eq. (3.2) and Eq. (3.3). As all the reuse vectors satisfy the conditions, array \(A\) is localized.

Next, we try to determine the tiling for array \(A\). Since all the access matrices are invertible, the data-space hyperplanes obtained through all the references are:

\[
H'_s = \{(1, 1) \cdot Q^{-1}_{a_1}, (1, 1) \cdot Q^{-1}_{a_2}\} = \{(1, 1)\},
\]

and the data-time hyperplanes are:

\[
H'_t = \{(0, 1) \cdot Q^{-1}_{a_1}, (0, 1) \cdot Q^{-1}_{a_2}\} = \{(0, 1), (1, 0)\}.
\]

For these candidate hyperplanes, we then use Eq. (3.4) to decide the final data-space and data-time hyperplanes for tiling, which are \((1, 1)\) (with \(d_r = 1\)) and \((0, 1)\) (with \(d_r = 16\)), respectively. The transformation matrix \(H\) is \(((0, -1/16), (1/4, 1/4))^T\) and the transformation function \(f\) is:

\[
f(\vec{a}) = \begin{pmatrix}
\vec{a} - \begin{bmatrix}
0 & -1/16 \\
1/4 & 1/4
\end{bmatrix} \cdot \vec{a}
\end{pmatrix} - \begin{bmatrix}
16 & 4 \\
-16 & 0
\end{bmatrix} \cdot \begin{bmatrix}
0 & -1/16 \\
1/4 & 1/4
\end{bmatrix} \cdot \vec{a}.
\]

### 3.4.4 Cache-Hierarchy Aware Memory Mapping

This step maps the tiles obtained from the previous step in the data space to the linear memory space based on the cache hierarchy in the target multicore architecture. Our goal is to reduce the memory footprint of disjoint working sets of multiple threads at any given time, such that capacity misses and inter-thread conflicts in shared caches are minimized. It is worth pointing out that, by using tiles as mapping unit, we can pack the locally accessed data elements into consecutive memory locations, which reduces TLB misses (such as a miss when going from \(A[i][j]\) to \(A[i + 1][j]\)) and improves the effectiveness of hardware prefetching as
To make our explanation easy to follow, we employ the access pattern for the two-dimensional array shown in Figure 3.1 and the cache hierarchy shown in Figure 1.1(c) to describe our mapping strategy, which is essentially a process of constructing customized memory patterns according to the given cache hierarchy, and carried out from the first level caches to the last level caches in a top-down fashion. Specifically, we first construct the pattern \( <T_1, T_2> \) of size \( L_1 \) by filling a memory chunk of size \( L_1/2 \) with tiles/data from thread \( T_1 \) and another memory chunk of size \( L_1/2 \) with tiles/data from thread \( T_2 \). The reason for this is that, ideally, we would like \( T_1 \) and \( T_2 \) each to utilize half of the shared L1 space for this localized array access, and the separate memory layout helps reduce the conflicts between \( T_1 \) and \( T_2 \) at runtime, especially when the L1 cache is fully-associative.

Next, we construct the pattern \( <T_3, T_4> \) of size \( L_1 \) for threads \( T_3 \) and \( T_4 \) in a similar way, and use patterns \( <T_1, T_2> \) and \( <T_3, T_4> \) to form the pattern \( <T_1, T_2, T_3, T_4> \) of size \( L_2 \) as follows: repeat the pattern \( <T_1, T_2> \) until half of \( L_2 \) size and then change to the pattern \( <T_3, T_4> \) for the rest. The motivation behind this is that, we want to alleviate the space contention among threads \( T_1, T_2, T_3 \) and \( T_4 \) in the shared L2 cache. Third, we construct the pattern \( <T_5, T_6, T_7, T_8> \) of size \( L_2 \) for threads \( T_5, T_6, T_7 \) and \( T_8 \), and use it together with the pattern \( <T_1, T_2, T_3, T_4> \) to form the pattern \( <T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8> \) of size \( L_3 \), as

Figure 3.5: Cache hierarchy-aware memory mapping for examples in Figure 1.1(c) and Figure 3.1.
shown in Figure 3.5. Finally, we map the entire array to the linear memory space using the pattern \(< T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8 >\) repeatedly. Note that, the data in the non-contiguous memory chunks assigned to a particular thread, such as the blocks marked with \(T_1\) in Figure 3.5, are stored sequentially as determined by our data-time hyperplanes. At this point, one can make the following observations: (i) each memory pattern is constructed based on a cache in the hierarchy, and it encodes the corresponding cache’s thread sharing information as well as size and connectivity with a subset of upper level caches; therefore, the memory pattern of the last level cache reflects the thread sharing and connectivity of all levels of caches; and (ii) the constructed patterns conform to the characteristics of a localized array in both time and space dimensions.

We now discuss the address calculation for this mapping strategy, the goal of which is to find the starting address of each memory chunk assigned to a thread. Since data of different threads are stored independently, we focus on the address calculation for one thread, and different threads only differ in the beginning address of their first memory chunk, i.e, the base address. The key insight is that, the memory spaces occupied by a thread depend on the connectivity and sizes of caches that this thread accesses, which have already been captured by the memory patterns that exhibited by this thread. Consider \(T_1\) in Figure 3.5 for example. Assuming that its base address is \(B_1\), the starting address of its \(x^{th}\) memory chunk (\(x\) starts from 0) can be calculated as \(B_1 + A_3 + A_2 + A_1\), where \(A_3, A_2\) and \(A_1\) denote the respective starting addresses of patterns \(< T_1, T_2, T_3, T_4 >\), \(< T_1, T_2, T_3, T_4 >\) and \(< T_1, T_2 >\) that its \(x^{th}\) memory chunk belongs to. If pattern \(< T_1, T_2 >\) repeats \(C_1\) times in pattern \(< T_1, T_2, T_3, T_4 >\), which means \(L_2/2 = C_1 L_1\), and pattern \(< T_1, T_2, T_3, T_4 >\) repeats \(C_2\) times in pattern \(< T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8 >\), i.e., \(L_3/2 = C_2 L_2\), then we have \(A_1 = (x%C_1)L_1, A_2 = ((x/C_1)%C_2)L_2, A_3 = (x/(C_1C_2))L_3\), where \(L_1, L_2\) and \(L_3\) are the corresponding pattern (cache) sizes. Similarly, the starting address of the \(x^{th}\) memory chunk of thread \(T_2\) can be formulated as \(B_2 + A_3 + A_2 + A_1\), where \(B_2\) is the base address of \(T_2\). In general, for a symmetric cache hierarchy with evenly-distributed threads across the cores, let \(n\) be the number of levels, \(L_i\) be the cache size at the \(i^{th}\) level, \(M_i\) be the number of caches that a level-\(i\) cache connects to at level \((i-1)\) (e.g., \(M_2 = 2\) in Figure 3.5, since an L2 cache is connected to two L1 caches), \(p\)
be the number of threads per L1 cache, and $C_i$ represent the times a level-$i$ pattern (corresponding to a level-$i$ cache) repeats inside a level-$(i + 1)$ pattern. The starting address of the $x^{th}$ memory chunk of thread $T_t$ can then be calculated as:

$$
\begin{align*}
B_t + A_n + A_{n-1} + \cdots + A_2 + A_1, \\
A_1 &= (x\%C_1)L_1, \\
A_i &= ((x/(C_1\cdots C_{i-1}))\%C_i)L_i, \quad 2 \leq i \leq n-1, \\
A_n &= (x/(C_1\cdots C_{n-1}))L_n, \\
C_k &= L_{k+1}/(M_{k+1}L_k), \quad 1 \leq k \leq n-1,
\end{align*}
$$

where $B_t$ is the base address of thread $T_t$. Note that, $C_n$ is equal to $D/L_n$, with $D$ representing the size of the localized array, and each memory chunk has a size of $L_1/p$.

Observe that, Eq. (3.5) essentially gives the index calculation for linearizing an $n$-dimensional array, the $k$th dimension of which has a length of $C_k$. In a sense, our approach creates a virtual multi-dimensional memory space for each localized array, where the number of dimensions is equal to the number of cache levels in the hierarchy. By transforming the original linear memory space to a virtual multi-dimensional memory space, we can bound the memory usage of threads at runtime through limited lengths of dimensions. The pseudo-code of our data layout transformation is given in Algorithm 2. To reduce index calculation overheads, we employed several low-level optimization techniques similar to those proposed in [37].

### 3.5 Experimental Evaluation

#### 3.5.1 Setup

We evaluated our data layout optimization scheme using the benchmark programs from the SPECOMP [28], SPLASH-2 [38], and PARSEC [39] suites as well as several scientific and DSP applications on an Intel quad-core machine. In this architecture (see Table I), each core has private L1 data and instruction caches, and each pair of cores share an on-chip L2 cache. Table 5.2 lists the benchmark programs used in our evaluation. Lattice, bit reverse, convolution, block FIR, and
Algorithm 2 Transformed data space to linearized memory address mapping.

1: **INPUT**: $k$ threads; transformed $(m+n-1)$-dimensional array $B$; cache hierarchy with $s$ level caches; memory starting address $M$; $p$ is the number of threads per L1 cache;

2: **OUTPUT**: Memory layout for array $B$.

3: for $t = 0 \rightarrow k - 1$ do

4:   Initialize $B_t$; $M_t \leftarrow B_t$; $x_t \leftarrow 0$;

5: end for

6: for $a_1 = 0 \rightarrow d_1 - 1$ do

7:   ...

8: for $a_{m+n-2} = 0 \rightarrow d_{m+n-2} - 1$ do

9:   for $t = 0 \rightarrow k - 1$ do

10:      if $B(t,a_1,a_2,\ldots,a_{m+n-2}) \neq \text{NULL}$ then

11:         $Mem[B(t,a_1,a_2,\ldots,a_{m+n-2})] \leftarrow M_t$;

12:      if $\text{counter} == L_1/p$ then

13:         $x_t++$;

14:         $A_1 \leftarrow (x_t\%C_1)L_1$;

15:      :

16:         $A_i \leftarrow ((x_t/(C_1\cdots C_{i-1}))\%C_i)L_i$, $2 \leq i \leq s - 1$;

17:         $A_s \leftarrow (x_t/(C_1\cdots C_{s-1}))L_s$;

18:         $M_t \leftarrow B_t + A_s + A_{s-1} \ldots + A_1$;

19:         $\text{counter} \leftarrow 0$;

20:      else

21:         $M_t++$; $\text{counter}++$;

22:      end if

23: end if

24: end for

25: end for

26: ...

27: end for

28: Align all the data stored in the memory to form a continuous memory address.

complex FFT are in-house DSP applications, and Triangular and Gauss-Seidel are two widely-used scientific solvers. For the benchmarks from SPECOM, SPLASH-2 and PARSEC, we used the largest input data sets available to us, and the data set sizes for the remaining benchmarks were between 85.1 MB and 12.8 GB.

The last level (L2) cache miss rates and parallel execution times for our original benchmark programs, collected using PerfMon [40], are given in Figures 3.6 and 3.7, respectively. It can be seen from Figure 3.6 that we cover a large spectrum
Table 3.1: Intel quad-core configuration.

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Type</td>
<td>Xeon X3380 (clocked at 3.17GHz)</td>
</tr>
<tr>
<td>System Bus Frequency</td>
<td>1.3GHz</td>
</tr>
<tr>
<td>L1</td>
<td>32KB/core; 8-way; 2 cycle latency</td>
</tr>
<tr>
<td>L2</td>
<td>total 12MB (2 × 6MB); 24-way, 15 cycle latency</td>
</tr>
<tr>
<td>Off-Chip Memory</td>
<td>~ 320 cycles</td>
</tr>
</tbody>
</table>

Table 3.2: Benchmarks.

<table>
<thead>
<tr>
<th>SPECOMP</th>
<th>wupwise, swim, gafort, applu, galgel, equake, apsi, mgrid, fma3d, art, ammp</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPLASH-2</td>
<td>barnes, cholesky, radix, ocean, radiosity, fmm, volrend</td>
</tr>
<tr>
<td>PARSEC</td>
<td>bodytrack, canmeal, facesim, freqmine, streamcluster</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>lattice, bit reverse, convolution, block FIR, complex FFT, Triangular, Gauss-Seidel</td>
</tr>
</tbody>
</table>

Figure 3.6: L2 cache miss rates of our original benchmarks.

Figure 3.7: Execution times of our original benchmarks.

of benchmarks as far as diversity of their L2 behavior is concerned. In particular, L2 miss rates vary between 6.9% and 39.8%. As stated earlier, a unique characteristic of our proposed scheme is that it determines memory layouts of arrays.
considering the cache hierarchy of the target multicore architecture. To evaluate its advantages against existing data locality optimization strategies proposed by prior research, we implemented and tested the following schemes on our benchmark programs. In naming these schemes, we use “def” to indicate our default iteration distribution strategy where, for \( P \) threads, iteration space is divided into \( P \) chunks (each chunk contains a set of consecutive iterations) and each thread is assigned a chunk. In comparison, we use “opt” to denote a locality-aware distribution of iterations to threads, where we divide iteration space into small-sized chunks (a total of \( M \) chunks where \( M > P \)), and if two chunks share data, they are allocated to two threads that share an L2 cache (under a given thread-to-core mapping). A load-balancing constraint ensures that each thread gets the same number of chunks. Note that “def” and “opt” do not differ from one another in
how loops are parallelized; the only difference between them is how loop iterations are distributed among threads.

- **L**(def). This scheme implements a suite of well-known data locality optimizations that restructure loop nests. It targets a single loop nest at a time and reorganizes the order in which loop iterations are executed using loop transformations such as loop permutation, iteration space tiling and scaling.

- **D**(def). This scheme employs the memory layout optimization proposed by [4], which considers the data access pattern of the loop nest, and determines an ideal memory layout for each multi-dimensional array. In some cases, the resulting layout may be different from any dimension reordering. This version in a sense represents the state-of-the-art data layout optimization (we also performed experiments with [41]; the output codes generated by [4] and [41] were very similar).

- **hierarchical**(def). This scheme implements our proposed hierarchical inter-
thread memory layout optimization.

$L^{(\text{opt})}$, $D^{(\text{opt})}$, and $\text{hierarchical}^{(\text{opt})}$ are the same as $L^{(\text{def})}$, $D^{(\text{def})}$ and $\text{hierarchical}^{(\text{def})}$, respectively, except that they are applied after optimized computation distribution explained above. In addition to these schemes, we also experimented with original$^{(\text{opt})}$, which corresponds to optimized computation distribution of the original codes without any specific data locality optimization for cache performance. Unless stated otherwise, all cache miss and execution time results presented below are with respect to the original versions of the applications, denoted as original$^{(\text{def})}$. The schemes listed above are implemented in the SUIF2 compiler [27]. For all schemes (including the original codes), the transformed source codes are compiled using the native compiler on our multicore machine with the -O3 optimization level. That is, all versions use the same set of low-level optimizations.

3.5.2 Results

Figure 3.8 plots the improvements (reductions) in the cache misses experienced by the last level cache of our multicore architecture when the default computation distribution is employed. Our first observation is that we can roughly divide the benchmarks into two groups: ones that benefit from locality optimization and ones that do not. For example, applications such as gafort, galgel, art, volrend and triangular get very little benefit from data locality optimization whether one uses optimized computation distribution or not. The reason why our hierarchical scheme is not successful with these programs is because it has little opportunity to apply (i.e., either most of the references were non-affine or the condition testing for localization failed most of the time). When we look at the results with the remaining benchmarks, however, it can be seen that our hierarchical layout optimization generates better results than conventional loop and data layout optimization schemes in most cases. Overall, with respect to original$^{(\text{def})}$, the average savings achieved by L$^{(\text{def})}$, D$^{(\text{def})}$, and hierarchical$^{(\text{def})}$ are 2.6%, 3.7%, and 11.2%, respectively. That is, while prior data layout optimizations (D$^{(\text{def})}$) does not perform very well in the multicore context, our hierarchy-aware layout optimization performs very well.
We now study the results (see Figure 3.9) with optimized computation distribution. While the magnitudes of the savings observed with these schemes are higher compared to those in Figure 3.8, we see that the trends among different schemes are similar. As in the default mapping case, hierarchical(opt) outperforms all remaining schemes in most benchmarks. When we do a cross-comparison between Figure 3.8 and Figure 3.9, we see that benchmarks such as barnes, cholesky, lattice and complex FFT get significant boost from optimized computation placement, indicating that, for these benchmarks, careful loop iteration-to-thread assignment can be critical. However, in most of the remaining benchmarks, the benefits are more modest. The main takeaway message from the results in Figures 3.8 and 3.9 is that the cache hierarchy-aware data layout optimization works well whether optimized computation distribution is used or not. Although not presented here, we also want to mention that hierarchical(def) and hierarchical(opt) reduced data TLB misses by 2.1% and 2.9%, respectively.

Figures 3.10 and 3.11 show the reductions in execution times with respect to original(def). While these savings are lower than the improvements in last level cache statistics (as expected), they are still significant. The average improvements brought by hierarchical(def) and hierarchical(opt) are around 8% and 12.5%, respectively.
Off-Chip Data Locality Optimization

4.1 Motivational Results

Most of the prior multicore related work on NoC optimizations focus exclusively on on-chip cache accesses [12, 13, 42, 43, 44]. However, depending on the data intensity and reuse patterns of an application, off-chip accesses can also play an important role in shaping overall performance characteristics. To illustrate this, we present in Figure 4.1a (the first bar for each benchmark) the contribution of off-chip data accesses to the total data accesses for a set of 12 multithreaded applications in an $8 \times 8$ mesh-based multicore system with private L2s (details of our experimental platform will be given later). These results indicate that off-chip accesses can contribute, on an average, to about 22.1% of all data accesses. The second bar for each benchmark in Figure 4.1a on the other hand plots the percentage increase in the network latency of on-chip accesses (cache accesses) as a result of off-chip accesses. This increase occurs because the on-chip accesses and off-chip accesses contend for the same NoC resources (e.g., message buffers, links). We observe the increase ranges between 10.5% and 28.5% (averaging on 20.1%), indicating that off-chip accesses can cause significant (additional) network latency for on-chip (cache) accesses. Finally, the third bars in Figure 4.1a represent the time spent, on an average, by an off-chip request on the network as a fraction of the total off-chip data access time (network time plus memory controller stall time plus memory access time). We see that network latency can contribute up to 28.3% of total off-chip data access latency. There are three factors that contribute
to the latency experienced by an off-chip data request in the on-chip network: (1) base network latency of the request, which is a function of the distance between the requesting core and the target memory controller (this latency is incurred even if no network contention takes place), (2) interferences from other off-chip accesses, and (3) interferences from on-chip accesses. Figure 4.1b shows the breakdown of the time spent by off-chip requests in the on-chip network into these three components. We see from this plot that most of the overheads that occur on top of the base network latency (of off-chip requests) are due to interferences caused by on-chip accesses. Overall, the results presented in Figures 4.1a and 4.1b show that off-chip data accesses can (1) slow down on-chip data (cache) accesses significantly, (2) spend significant amount of time in the network, and (3) get delayed due to interferences from on-chip accesses. We observed similar numbers in the case of shared L2 as well. Therefore, it can be very useful in practice to optimize off-chip data accesses in an NoC-based multicore system.
4.2 Problem of Off-chip Accesses

The inherent problem that causes the remote off-chip accesses is the mismatch between the Data-to-Core mapping provided by the application and the underlying Data-to-MC (memory controller) mapping imposed by the hardware.

**Data-to-Core mapping:** There are many ways to parallelize a loop-intensive application. For example, in OpenMP’s static scheduling, the computations are evenly divided into contiguous chunks and assigned to the cores in order. An application parallelized in this way indirectly indicates the portion of data elements that will be accessed by each core, which is called the “Data-to-Core mapping” in this paper.

**Data-to-MC mapping:** The underlying Data-to-MC mapping tells how physical addresses are distributed across memory controllers. In a system with $N$ memory controllers, $\log(N)$ bits of the physical address are used to determine the mapping of data to memory controllers. Depending on where these bits are taken from the physical address, different data interleavings (physical address-to-memory controller mapping) can be implemented in hardware. Figure 5.8 shows the virtual-to-physical address translation and the usage of physical address bits in the target architecture. There are two alternative virtual-to-physical address translations we consider in this work. In the first one, the first $\log(N)$ bits following the cache block (line) offset bits are used, which results in cache block interleaving across memory controllers. In other words, if a cache block with address $B_i$ is mapped to MC $j$, the next cache block at address $B_{i+1}$ will be mapped to MC $j + 1$. This can be expressed as $j = B_i \mod N$. In this cache-block interleaving, the bits used for memory controller selection are not modified by the virtual-to-physical address translation process. Therefore, one can statically determine the memory controller where a particular data element will reside by simply examining its virtual address. Alternatively, memory controller selection bits can be taken from the first bits after the page offset field. In this case, the granularity of interleaving will be a page such that two consecutive pages in the physical address space will be distributed to two different memory controllers.

Consider the private L2 case shown in Figure 1.3a as an example. A Data-to-Core mapping for a set of data elements stored in contiguous memory locations is
Figure 4.2: Virtual-to-physical address translation and the interpretation of the physical address bits.

Figure 4.3: Using data layout transformation to localize off-chip memory accesses. Illustrated in Figure 4.3, where the underlying Data-to-MC mapping is also given. In the original layout, the off-chip access requests to the data elements mapped to Core 0 will be sent to all four memory controllers, while the desired memory controller for these accesses should be MC1. That is to say, ideally, all the off-chip accesses to the data elements mapped to a core should be sent to the memory controller that is the closest to this core, but the underlying Data-to-MC mapping prevents this to be realized. One way to solve this problem is to rearrange/reorder data elements on the virtual address space (called data layout transformation) such that each data element can be mapped to the desired memory controller. For example, the transformed layout at the bottom of Figure 4.3 successfully eliminates the mismatch mentioned above: the data elements accessed by Core 0 are always mapped to MC1. How to generate this kind of layout is the main task of our proposed framework.
Figure 4.4: Different L2-to-MC mappings. The data accessed by cores that belong to a cluster are allocated/accessed from the corresponding controllers (indicated by arrows).

4.3 Framework Overview

Our compiler-guided off-chip access localization framework consists of two steps. In the first step, which is called Determining Data-to-Core Mapping, we identify the data/data regions accessed by each core/thread locally, and determine the mapping between data elements and cores. In the next step, which is called Layout Customization, we reorganize the layout of the data elements accessed by each core on the virtual address space such that the off-chip access requests of these data elements can always be sent to the nearby/desired memory controller to the L2 bank that emits these requests.

To define the “desired” memory controllers, we employ a concept called L2-to-MC mapping which is considered as an input for the second step and provided by the user/programmer. Figure 4.4 shows two sample L2-to-MC mappings with four memory controllers. The nodes covered by the same shaded area form a cluster. All the off-chip access requests from the L2s connected to the nodes in the same cluster will be sent to the same set of memory controllers. Each of the L2-to-MC mappings illustrated in Figures 4.4(a) and (b) has its own advantages and drawbacks. Specifically, in (a), since all the off-chip access requests from the same

Note that, not any L2-to-MC mapping is valid. In order to obtain a desired data layout, we require, first, each cluster must contain an equal number of cores, and second, each cluster should be assigned to an equal number of memory controllers. This is due to the strip-mining and permutation transformations that will be introduced later.
cluster will always be sent to the nearest memory controller, the average distance that these requests need to travel is less than the case in (b); that is, (a) localizes off-chip accesses better than (b). However, if too many off-chip access requests are sent by cores to the same memory controller simultaneously, the time required to process these requests can significantly degrade application performance. In this case, (b) could be a better choice since now the requests sent from one core will be processed by two memory controllers. This reduces the pressure on each memory controller to some extent and, therefore, helps with processing of off-chip accesses. In other words, (b) can enjoy better memory level parallelism (MLP) in processing the off-chip accesses. To summarize, different L2-to-MC mappings exhibit different locality vs. parallelism tradeoffs. The user can specify any L2-to-MC mapping as long as the specified mapping satisfies the two constraints discussed above. Our approach will then try to localize the off-chip accesses based on this L2-to-MC mapping.

In the rest of our discussion, we mainly focus on the case where the underlying Data-to-MC mapping is on the cache-block interleaving granularity. In Section 4.4.4 we will discuss how to handle the case where page interleaving is employed.

4.4 Off-chip Access Localization

4.4.1 Background

The iteration space of an m-level loop nest can be viewed as an m-dimensional polyhedron bounded by the loop bounds. Each iteration (each point in this polyhedron) can be expressed by an iteration vector $\vec{i} = (i_1, i_2, \cdots, i_m)^T$, where $i_1, i_2, \cdots, i_m$ are the loop iterators. Similarly, the data space of an n-dimensional array can be viewed as an n-dimensional polyhedron bounded by the array bounds. Each data element (each point in this polyhedron) can be expressed by a data vector $\vec{a} = (a_1, a_2, \cdots, a_n)^T$, where $a_1, a_2, \cdots, a_n$ are the array indices. The mapping between iteration space and data space is represented by the array references, which can be written as $\vec{r} = A\vec{i} + \vec{\sigma}$, where $A$ is an $n \times m$ constant matrix called the access matrix, and $\vec{\sigma}$ is an $n \times 1$ constant vector. For example, the reference
Figure 4.5: The original and transformed data space for a set of data elements accessed by 64 cores.

\[ A[i_1][2i_2 + 1] \] in a two-level loop nest (with loop iterators \( i_1 \) and \( i_2 \)) can be expressed as:

\[
\vec{a} = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \cdot \vec{i} + \begin{pmatrix} 0 \\ 1 \end{pmatrix},
\]

where \( \vec{i} = (i_1, i_2)^T \) and \( \vec{a} = (a_1, a_2)^T \).

A hyperplane \( h \) in an \( k \)-dimensional polyhedron is a flat subset of \((k - 1)\) dimensions, which can be characterized by a \( k \times 1 \) vector \( \vec{h} = (h_1, h_2, \ldots, h_k) \) and a constant \( c \). In our context, \( \vec{h} \) is called the hyperplane vector and \( c \) is called the hyperplane offset. Any point \( \vec{p} = (p_1, p_2, \ldots, p_k)^T \) on \( h \) satisfies \( \vec{h} \cdot \vec{p} = c \).

In this work, we focus on parallelized affine loop nests where the array subscript expressions and loop bounds are affine functions of enclosing loop indices and loop-independent variables. One of the frequently-used loop parallelization and distribution schemes in HPC codes is called block-cyclic distribution, where, with one thread per core, an \( m \)-dimensional iteration space is evenly partitioned into chunks (the last chunk may have a smaller number of iterations) by \( w \) set of parallel hyperplanes. For simplicity, in the rest of our discussion, we assume \( w=1 \), i.e., there is only one set of parallel hyperplanes that is orthogonal to the \( u \)-th dimension of the iteration space.\(^2\) Therefore, the hyperplane vector that represents this set of parallel hyperplanes is a \( 1 \times m \) unit vector (denoted as \( \vec{h}_I \)) in the form of

\[
\begin{pmatrix}
0, 0, \ldots, 0, 1, 0, \ldots, 0
\end{pmatrix}_{u-1}.
\]

\(^2\) The mathematical expressions/equations derived under this assumption can be easily extended to the multi-dimensional loop parallelization case.
4.4.2 Determining Data-to-Core Mapping

In this step, we isolate the data elements touched by different threads/cores. Specifically, we evenly partition the data space into data blocks by a set of parallel hyperplanes orthogonal to a given dimension, such that, most of the data elements in a given data block are accessed by the same thread/core. This dimension is called the data partitioning dimension. If we specify the \( v \)-th dimension as the data partitioning dimension, then the set of parallel hyperplanes that partition the transformed data space can be represented by the hyperplane vector \( \vec{h}_A \), which is a \( 1 \times n \) unit vector in the form of \((0,0,\cdots,0,1,0,\cdots,0)\). Figure 4.5 illustrates the original and transformed data spaces for a set of data elements accessed by 64 cores, where the data partitioning dimension is specified as the slowest-varying dimension, i.e., \( v = 2 \). It should be noted that, in the original data space, although most data elements in the same data block are accessed by the same thread/core, these data blocks are not formed by the set of parallel hyperplanes orthogonal to the \( v \)-th dimension. Therefore, they do not form a valid Data-to-Core mapping.

We employ the unimodular data transformation to find such partitioning, which can be characterized by a transformation matrix \( U \) [2]. Each data vector \( \vec{a} \) in the original data space is mapped to a unique vector \( \vec{a}' \) in the transformed data space, i.e., \( \vec{a}' = U\vec{a} \), and the array reference \( \vec{r} \) is changed accordingly to \( \vec{r}' \), i.e., \( \vec{r}' = U\vec{r} \).

Now, let us first discuss, in cases where there is only one array reference \( \vec{r} \), how to obtain the transformed data space by determining the entries of the transformation matrix \( U \). Any two iterations \( \vec{i}_1 \) and \( \vec{i}_2 \) that reside on a hyperplane defined by a hyperplane vector \( \vec{h}_I \) should satisfy:

\[
\vec{h}_I(\vec{i}_1 - \vec{i}_2) = 0. \tag{4.1}
\]

Let \( \vec{i}_{1,2} = \vec{i}_1 - \vec{i}_2 \), and \( \vec{e}_i \) be an \( m \times 1 \) unit vector, where 1 appears at the \( i \)-th position; the solution set for \( \vec{i}_{1,2} \) can be expressed as \( \sum_{i=1,i\neq u}^{m} k_i \vec{e}_i \), where \( k_i \) is an arbitrary integer. Similarly, in the transformed data space, the two data elements \( \vec{a}'_1 \) and \( \vec{a}'_2 \) accessed by these two iterations through \( \vec{r}' \) should always reside on the same hyperplane defined by the hyperplane vector \( \vec{h}_A \). Therefore, we have

\[3\]In an attempt to reduce the padding overhead mentioned in Section 4.4.3, this dimension is always chosen to be the slowest-varying dimension (e.g., first dimension in a row-major layout).
\( \vec{h}_A (\vec{a}_1' - \vec{a}_2') = 0 \). Assuming \( \vec{r} = A\vec{r} + \vec{\sigma} \) and \( \vec{\sigma}' = U\vec{a} \), we further have:

\[
\vec{g}_v A (\vec{i}_1 - \vec{i}_2) = 0, \tag{4.2}
\]

where \( \vec{g}_v \) is the \( v \)-th row vector of \( U \). In other words, any two iterations \( \vec{i}_1 \) and \( \vec{i}_2 \) that satisfy Eq. (4.1) must also satisfy Eq. (4.2). As a result, we have \( \vec{g}_v A\vec{e}_i = 0 \), where \( 1 \leq i \leq m, \ i \neq u \). Let \( B \) be a matrix that consists of all the column vectors of \( A \) except the \( u \)-th one (called a submatrix of \( A \)). Then, the last expression can be re-written as:

\[
B^T \vec{g}_v^T = 0. \tag{4.3}
\]

The above homogeneous linear system can be solved by Integer Gaussian Elimination [45]. This also indicates that the desired transformation matrix \( U \) is completely determined by its row vector \( \vec{g}_v \). In cases where the solver returns a non-trivial solution for \( \vec{g}_v \), we determine the remaining \( n - 1 \) row vectors such that \( U \) is unimodular.

Next, we study how to determine \( U \) in a global sense when there are multiple references to an array in different parallelized loop nests. Assuming that there are \( k \) different submatrices \( B_1, B_2, \cdots, B_k \) for these references, based on Eq. (4.3), we have \( k \) homogeneous linear systems to solve, each corresponding to a submatrix. A unique \( \vec{g}_v \) that satisfies all these systems may or may not exist. That is, we may not always end up in a transformed data space where all the data elements on the hyperplane orthogonal to \( \vec{h}_A \) are accessed by a single thread. To address this potential problem, our strategy is to assign a weight to each submatrix to determine the most “beneficial” linear system that, when solved, satisfies the majority of references. Specifically, assuming that there are \( s \) references in the given set of loop nests that have the same submatrix \( B_i \), then the weight of \( B_i \), denoted as \( W(B_i) \), is the total number of occurrences of these references, i.e., \( W(B_i) = \sum_{j=1}^{s} n_j \), where \( n_j \) is estimated by the product of the trip counts (the number of iterations) of the loops that enclose the said reference.
4.4.3 Layout Customization

So far we have determined the Data-to-Core mapping and assumed that the user provides a valid L2-to-MC mapping. If it is the local L2 cache that sends the off-chip access request to the memory controller, which is true for the case of private L2s, then these two mappings actually indicate a desired Data-to-MC mapping. However, because of the underlying Data-to-MC mapping mechanism, this desired mapping may not always be realized directly. We address this problem (of implementing the desired Data-to-MC mapping) by customizing the layout obtained in Section 4.4.2 according to the L2-to-MC mapping specified by the user.

To do this, we employ two data transformation techniques, strip-mining and permutation, originally proposed in [46], for an entirely different purpose. Let \( N_i \) denote the size of the array along the \( i \)-th dimension. Then, strip-mining with a block size of \( s \) transforms this dimension into two dimensions with the size of \( N_i/s \) and \( s \); and a reference \( \vec{r} = (\cdots, r_i, \cdots)^T \) becomes \( (\cdots, r_i/s, r_i \% s, \cdots)^T \). After this transformation, \( r_i/s \) and \( r_i \% s \) can be used to identify the index of the block and the offset within a block, respectively. In comparison, permutation switches the positions of two dimensions in an array to change the data placement in the linear memory space; and a reference \( \vec{r} = (\cdots, r_i, \cdots, r_j, \cdots)^T \) becomes \( (\cdots, r_j, \cdots, r_i, \cdots)^T \). The run-time overhead associated with these transformations comes mainly from the division and module operations employed. In our current implementation, we use the techniques proposed in [46] and [12] to reduce these costs. We also employ padding [3] to keep the base addresses of arrays aligned to the desired memory controller, and align data elements within an array to make the strip-mined dimension divisible by \( s \).

Next, we discuss how to use above techniques to customize the data layout for private L2s and shared L2.

---

4As will be shown later, one can use layout transformation techniques to make this true for the shared L2 case as well.
4.4.3.1 Private L2s

For a multicore system with private L2 caches, our first step is to transform original reference (obtained from Section 4.4.2) \((\cdots, r_v, \cdots, r_n)^T\) to \((\cdots, R(r_v), \cdots, r_n)^T\), where the \(v\)-th dimension is the data partitioning dimension identified in Section 4.4.2, and \(R(r_v)\) is a set of transformed array indices that can be used to identify the cluster for each data element accessed through this reference. Recall that, in order to identify the data blocks touched by each core/thread, the data space is partitioned by a set of parallel hyperplanes orthogonal to this dimension. Therefore, \(R(r_v)\) can be obtained by performing multiple strip-minings on the \(v\)-th and the newly-generated dimensions. Specifically, assuming that we have \(c_x \times c_y\) number of clusters and each cluster has \(n_x \times n_y\) number of cores, then \(R(r_v)\) can be expressed as \(((r_v/b)/(n_y \times c_y \times n_x))\%c_x, ((r_v/b)/n_y)\%c_y\), where \(b\) is the data block size, and the subscript “x” and “y” indicate the number along the X-axis and Y-axis, respectively.

Next, based on this transformed layout, we perform permutation and strip-mining on the fastest-varying dimension of the target array (e.g., the last dimension in a row-major layout) to obtain an interleaved layout. Assuming that \(k\) is
the number of memory controllers assigned to each cluster (see Figure 4.4 for sample assignments), and \( p \) is the cache block size (in terms of the number of data elements), by transforming the array reference into the form of \( (\cdots, r_n/(k*p), R(r_v), r_n\% (k*p))^T \), every consecutive \( k*p \) data elements will be accessed by the cores in the same cluster in a round-robin fashion, and the off-chip accesses made for \( k*p \) consecutive data elements will be sent to the memory controllers indicated by the L2-to-MC mapping.\(^5\) An example is given in Figure 4.6.

4.4.3.2 Shared L2

In a shared L2 based system [1], all the on-chip and off-chip requests for a data element are issued by its home (L2) bank, not by its local bank. The home bank for each data element is determined by the underlying Data-to-L2 Bank mapping mechanism, which is similar to the Data-to-MC mapping explained in Section 4.2. Therefore, localizing the on-chip accesses is also important for the application performance. However, for cache-block interleaving, this introduces an additional problem: if we only focus on improving the on-chip access locality, although the distance between the requester of the data and the L2 home bank that owns the data could be minimized, doing so could in turn degrade the off-chip access locality on the network, and vice versa. For example, the first \( p \) data elements are accessed by Core 0, and their home bank is the one that is connected to Core 0; the next \( p \) data elements are accessed by Core 1, and their home bank is the one that is connected to Core 1, and so on. From the on-chip accesses point of view, this is the ideal case since most of the data accesses become local. However, from the off-chip accesses point of view, this is problematic. Assuming that we have 4 memory controllers, then the first \( p \) data elements will be mapped to MC1, and the next \( p \) data elements will be mapping to MC2, and so on. Given the desired L2-to-MC shown in Figure 4.4(a), the off-chip accesses to the data elements whose home bank is connected to Core 2 will be sent to MC2 instead of the desired one – MC1.

One might wonder if there exists a layout such that both the on-chip and off-

\(^5\)We bind each thread to a core through a system call to ensure that the order of the cores is consistent with the order of memory controllers in the target two-dimensional grid (see Figure 1.3a).
chip accesses are localized. To show the difficulty of obtaining such a layout, let $addr(\vec{a})$ represent the virtual address of a data element $\vec{a}$. Then, the home bank that issues the off-chip accesses to $\vec{a}$, denoted as $id_{HB}$, can be expressed as:

$$id_{HB} = \frac{addr(\vec{a})}{p} \% N,$$

(4.4)

where $p$ is the cache block size and $N$ is the total number of cores on the network. Similarly, the memory controller to which the off-chip access request $\vec{a}$ is sent, denoted as $id_{MC}$, can be expressed as:

$$id_{MC} = \frac{addr(\vec{a})}{p} \% N',$$

(4.5)

where $N'$ is the total number of memory controllers. Based on Eqs. (4.4) and (4.5), we have $N \ast \delta_1 + id_{HB} = N' \ast \delta_2 + id_{MC} = \frac{addr(\vec{a})}{p}$. In the L2-to-MC mapping shown in Figure 4.4(a), we have $N = 64$, $N' = 4$, which indicates that $4 \ast (16 \ast \delta_1 - \delta_2) = id_{MC} - id_{HB}$, where $\delta_1$ and $\delta_2$ are integers. To satisfy this equation, $id_{MC}$ and $id_{HB}$ must be multiples of 4. However, most home banks and memory controllers may not satisfy this requirement. Therefore, it is really hard (in most cases) to find a layout localized for both on-chip and off-chip accesses.

To solve this problem, our strategy is to generate a localized layout for the on-chip accesses first, and then try our best to localize the off-chip accesses. Specifically, we first generate a layout with the property that most of the data accesses are local, i.e., home bank of each data block is the one connected to the core by which this data block is accessed. Similar to the case of the private L2 based system, we transform an array reference $(\cdots, r_v, \cdots, r_n)^T$ into the form of $(\cdots, \frac{r_n}{b}, R'(r_v), \frac{r_n}{p})^T$, where $R'(r_v)$ is a set of transformed indices that can be used to identify the L2 bank, and can be expressed as $R'(r_v) = (r_v/b) \% N$. After this transformation, every consecutive $p$ data elements will be accessed by the same core in a round-robin fashion. If we want to optimize the off-chip access under this new layout, one option we have is that, for a data element mapped to the memory controller far from its desired memory controller, instead of placing it into an address (memory location) where its on-chip access is localized while its

---

6One can also first generate the data layout localized for off-chip accesses and then try to localize the on-chip accesses as much as possible.
off-chip access is not, we can place it into an address such that, although neither of its on-ship access or off-chip access is local, both of its home bank and memory controller are close to the desired ones. Specifically, in the layout localized for on-chip accesses, if a data element resides on a memory address/location where the mapped memory controller is not even adjacent to the desired memory controller, then we skip this address and move that data element to the next closest memory address such that the mapped memory controller is adjacent to the desired memory controller, and all the data elements originally at or beyond this address will be moved forward accordingly. This requires us to replace the original reference $\vec{a} = (\cdots, r_n/p, R'(r_n), r_n\%p)^T$ with $\vec{a}'$ where: $\vec{a}' = \vec{a} + (0, \cdots, 0, \delta \cdot p)^T$. Here, $\delta$ is a counter that will be inserted at the loop level where the target array is accessed. It will be increased by 1 (i.e., $\delta = \delta + 1$) if $id_{MC}(\vec{a}) \in C$, where $id_{MC}(\vec{a})$ gives the desired memory controller for $\vec{a}$, and $C$ is the set of memory controllers that are not adjacent to this memory controller.

4.4.4 Extension to Page Interleaving

If the page-interleaving is adopted, one can still apply the data transformation scheme discussed so far by simply changing $p$ to the page size instead of cache block size. However, there is one additional problem that needs to be addressed: as opposed to cache block interleaving, when page interleaving is adopted, the bits used for memory controller selection are modified by the OS, which means that, it is the OS, instead of the compiler, that determines which memory controller a particular data element is mapped to. Therefore, the compiler-guided data layout transformation scheme discussed so far cannot directly enforce the desired Data-to-MC mapping. Instead, we need the OS to be involved. Figure 4.7 shows an example where, after data transformation, with an assist from the OS, the data elements accessed by the same set of cores (marked using the same texture) are assigned to the physical pages mapped to the same memory controller.\footnote{One could also achieve the desired Data-to-MC mapping without any data transformation. However, with the help from the compiler, the task for the OS becomes much easier.}

To realize the desired Data-to-MC mapping, we need to change the existing page allocation policy in the OS.

In most page allocation schemes, the OS maintains a free list, which holds
virtual pages that are not mapped into any physical address space. Whenever a running process needs a new memory space, the OS allocates free pages from this list and updates this list accordingly. In Solaris for example, when a page is put on the free list, it will be assigned a color. When a page is consumed from the free list, the virtual-to-physical algorithm gets the page from a color. New pages are allocated by calling the page_create_va() function. The page_create_va() function accepts the virtual address of the location to which the page is going to be mapped as an argument; then, the virtual-to-physical coloring algorithm can decide which color to take physical pages from.

To localize the off-chip accesses, we slightly change the page allocation algorithm (more precisely, the page_create_va() function) such that, when assigning a physical address space to a page with the given virtual address, we require the assigned physical address to belong to the desired memory controller. For example, in Figure 4.7, assuming that we only have two memory controllers, the data elements in the first memory chunk (page) should be mapped to the physical addresses whose Page-to-MC mapping bit (Figure 5.8) will be set to 0, and the data elements in the second memory chunk (page) should be mapped to the physical addresses whose Page-to-MC mapping bit will be set to 1. In other words, the modified page allocation algorithm assigns physical pages in a round-robin fashion to guarantee the desired distribution of pages across memory controllers.

At this point, there are two important issues that need to be clarified. First, the virtual memory layout generated by the first component of our proposed approach actually simplifies the modification to the page coloring algorithm. This is because, after the data transformation, we can easily obtain the desired Page-to-MC mapping bits for each data element based on the Data-to-MC mapping, which is indicated by the Data-to-Core mapping and L2-to-MC mapping. Second, if the memory space attached to the specified memory controller is full, an alternate memory controller is selected and the page is placed into the space managed by that controller. Consequently, our approach does not increase the number of page faults, i.e., the available physical memory space is fully utilized. We omit the discussion of how to identify the alternate memory controller, due to the lack of space.
virtual address space, original layout
virtual address space, after layout customization
physical address space, after OS page allocation

Figure 4.7: Changing Data-to-MC mapping through the OS support. In this example, we assume there are two memory controllers (MC1 and MC2).

Table 4.1: The simulated configuration.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cores and Caches</strong></td>
<td></td>
</tr>
<tr>
<td>Processor</td>
<td>two-issue, SPARC processor</td>
</tr>
<tr>
<td>Data/Instr. L1 Config.</td>
<td>16 KB (per node), 64 byte lines, 2 ways</td>
</tr>
<tr>
<td>L2 Config.</td>
<td>256KB (per node), 256 byte lines, 16 ways</td>
</tr>
<tr>
<td><strong>NoC</strong></td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td>8 × 8 two-dimensional mesh</td>
</tr>
<tr>
<td>Delays and Routing</td>
<td>16B links, 2-cycle pipeline, XY-routing</td>
</tr>
<tr>
<td><strong>Memory System</strong></td>
<td></td>
</tr>
<tr>
<td>Number of Memory Controllers</td>
<td>4 [same as page size]</td>
</tr>
<tr>
<td>Interleaving Granularity</td>
<td>4KB</td>
</tr>
<tr>
<td>Scheduling Policy</td>
<td>FR-FCFS [16]</td>
</tr>
<tr>
<td>Capacity</td>
<td>4GB</td>
</tr>
<tr>
<td>Device Parameters</td>
<td>Micron MT47H64M8 DDR2-800 timing parameters [47], 4 banks/device, 16384 rows/bank, 512 columns/row</td>
</tr>
<tr>
<td>Row Buffer Size</td>
<td>4KB [same as page size]</td>
</tr>
<tr>
<td></td>
<td>4 active row buffers per DIMM</td>
</tr>
<tr>
<td><strong>Optimization Parameters</strong></td>
<td></td>
</tr>
<tr>
<td>Interleaving Unit</td>
<td>4KB [same as page size]</td>
</tr>
<tr>
<td>L2-to-MC mapping</td>
<td>as shown in Figure 4.4(a)</td>
</tr>
</tbody>
</table>

### 4.5 Experiments

#### 4.5.1 Experimental Setup and Applications

The proposed data layout optimization is implemented using the SUIF infrastructure [27]. During the compilation process, first, a number of scalar optimizations is performed which include constant propagation, forward propagation, induction variable detection, constant folding, and scalar privatization. Next, unimodular loop transformations guided by array dependence analysis restructure the intermediate code for improving both parallelism and data locality (cache performance). Our proposed data transformation scheme is embedded as an additional pass after
these transformations. Finally, the code generator produces parallel code with calls to the parallel run-time library. All the experiments presented below are carried out using the GEM5 simulation environment [49], with our modified Linux kernel that implements the customized page allocation policy explained. The default system configuration used in our experiments is given in Table 4.1 (Figure 4.4(a) shows the high level view of our default system). We present results from a set of multithreaded programs from SPECOMP [28] and SPLASH2 suites [38], as well as five kernel computations. The description of these benchmarks are given in Table 5.2. The last column of Table 5.2 gives the L2 miss rates of our applications (for the configuration in Table 4.1). Although we use a 16-way set associative L2 cache, we observed that the conflict misses in L2 dominated. For the SPECOMP applications, we used the large input sets; and in SPLASH2 applications, we used the default input sets except in ocean and raytrace where we used 4 times larger input sets than advertised. The input sizes of the remaining applications ranged between 66.1MB and 2.2GB. In our 8 × 8 default configuration, all these applications run one thread per core and, in each experiment, we run only one multithreaded application. The original codes are compiled with the highest optimization level turned on. Also, unless otherwise stated, our default L2-to-MC mapping is the one shown in Figure 4.4(a), with private L2s (later we give a set of results with shared L2 as well).

\[\text{Table 4.2: Benchmarks used in our evaluation.}\]

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>L2 Miss Rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>swim</td>
<td>Weather prediction [28]</td>
<td>16.3</td>
</tr>
<tr>
<td>mgrid</td>
<td>Multi-grid solver in 3D potential field [28]</td>
<td>24.3</td>
</tr>
<tr>
<td>applu</td>
<td>Parabolic/elliptic partial differential equations [28]</td>
<td>27.1</td>
</tr>
<tr>
<td>apsi</td>
<td>Solves problems regarding temperature, wind, distribution of pollutants [28]</td>
<td>29.3</td>
</tr>
<tr>
<td>raytrace</td>
<td>Renders a three-dimensional scene onto a two-dimensional image plane [38]</td>
<td>17.8</td>
</tr>
<tr>
<td>ocean</td>
<td>Simulates large-scale ocean movements [38]</td>
<td>20.6</td>
</tr>
<tr>
<td>lu</td>
<td>LU decomposition [38]</td>
<td>14.8</td>
</tr>
<tr>
<td>mxmt</td>
<td>Matrix-matrix multiplication</td>
<td>33.8</td>
</tr>
<tr>
<td>3d-gs</td>
<td>A three-dimensional Gauss-Seidel method</td>
<td>29.9</td>
</tr>
<tr>
<td>mpeg4</td>
<td>AV digital data compression</td>
<td>14.4</td>
</tr>
<tr>
<td>filter-1</td>
<td>Implementation of Cauer filter [48]</td>
<td>24.4</td>
</tr>
<tr>
<td>filter-2</td>
<td>Another implementation of Cauer filter [48]</td>
<td>31.8</td>
</tr>
</tbody>
</table>

\[\text{We collected results with all benchmarks in SPLASH2 and SPECOMP, but present results for a smaller representative subset for clarity. The average execution time improvement brought by our approach is 17.2\% in the case of private L2s, and 19.3\% in the case of shared L2, when all SPECOMP and SPLASH2 codes are considered.}\]
Figure 4.8: Impact of off-chip access localization. The bars capture the accesses to MC1 in Figure 4.4a within an execution period. In the optimized case (b), most of accesses go to one memory controller (MC1), indicating that the off-chip accesses are localized.

4.5.2 Impact of Off-Chip Access Localization

We first present two representative maps to illustrate how our approach changes the off-chip traffic destined to a memory controller. Our objective is to see how much off-chip access localization our approach achieves in practice. For this experiment, we focus on one of our applications, apsi. For example, we have two graphs, Figures 4.8(a) and (b), for the controller MC1 shown in Figure 4.4(a). In both the graphs, the vertical axis plots the fraction of off-chip accesses made to this controller from each of our 64 nodes in an execution period. The coordinates of the nodes in our two-dimensional grid are given by the remaining two axes in Figure 4.8. The first graph is for the original case and the second one is for the case when using our proposed strategy. We also have similar graphs from other three controllers in our system (due to the space concern, we do not show them here).

The most important observation from these plots is that our approach localizes the off-chip data accesses successfully. That is, while in the original case the off-chip access requests to a memory controller come from all over the chip, in the optimized case, the accesses to the same memory controller are highly skewed towards the nearby cores.

Next, the impact of these localized accesses on application performance is quantified in Figure 4.9. The first bar in this graph plots the contributions of the off-chip accesses to the on-chip access latency. We see that the average value for this
parameter is about 12.1%, much lower compared to the original case (20.1%, as shown in Figure 4.1a). Similarly, the second bar for each benchmark gives contributions of network latency to overall data access latency. We observe about 9.5% reduction (compared to the original case) in the value of this parameter. Finally, the last bar plots the percentage reduction (improvement) in execution latencies. Our approach saves about 18.2% performance, when averaged across all programs.

4.5.3 Evaluation of Cache-Block Interleaving

We quantify in Figure 4.10 the performance benefits achieved when cache-block interleaving is adopted. All bars are normalized with respect to the case where the same granularity physical memory distribution is used without our approach. The first bar for each application in this graph gives contributions of off-chip accesses to on-chip access latency, and indicates that we achieve about 34.6% improvement. The second bar on the other hand gives the contributions of network latency to overall data access latency. We observe about 44.4% reduction in this parameter. Finally, the last bar plots the percentage improvement in execution latencies. Our approach saves about 20% performance when averaged across all applications.

4.5.4 Result with an Alternate L2-to-MC Mapping

We next present the results with different L2-to-MC mappings. Figure 4.11 gives the execution time improvement results with the mappings shown in Figure 4.4, where M1 and M2 in Figure 4.11 correspond to the mappings illustrated in Fig-
Reduction on Chip Access Latency
Reduction Memory Access Latency
Reduction in Execution Cycles

Figure 4.10: Application performance in the case that cache-block interleaving is adopted.

Figure 4.11: The execution time improvement results with the mappings shown in Figures 4.4(a) and (b).

ures 4.4(a) and (b), respectively (recall that M1 is our default mapping shown in Figure 4.4a). It can be seen that, in most of our applications, going from M1 to M2 results in a reduction in our savings. This is because in these applications, off-chip access localization is more important than high levels of memory-level parallelism (MLP), and the banks connected to an MC are able to satisfy the memory level parallelism demand of a cluster (of 16 cores). In 3d-gs, filter-1, and filter-2 however, going from M1 to M2 generated better results. This is because they exhibit much higher memory-level parallelism (MLP) demand compared to the remaining ones. As a result, one can sacrifice some off-chip access locality to meet this memory-level parallelism demand.

4.5.5 Sensitivity Analysis

Results with Different MC Placements. As pointed out in [50], modern flip-chip packaging allows sufficient escape paths from almost anywhere on the chip. In an NoC-based multicore, this enables designers to explore different place-
ments of memory controllers within the on-chip network. Recall that, so far in our experimental evaluation, we used the MC placement depicted in Figure 4.4(a). Figures 4.13(a) and (b) show two alternate memory controller placements (all with four MCs). The results (execution time improvements) with these placements are plotted in Figure 4.12 (P1, P2 and P3 correspond to the placements in Figure 4.4a, Figure 4.13a and Figure 4.13b, respectively). One can observe from these results that placement P2 generates slightly better results than others (an average improvement of about 20.7%), mainly because the average distance-to-controller is expected to be lower under this placement. We also observe however that our approach generates significant savings under all these MC placements.

**Results with Different MC Counts.** We next quantify the sensitivity to the number of memory controllers. To eliminate the effect from different memory controller placement, we increase the number of memory controllers based on the configuration depicted in Figure 4.4(a). These configurations are shown in Figure 4.14a and Figure 4.14b, and have L2-to-MC mappings similar to the one shown in Figure 4.4(a). The results with these new configurations are presented in Fig-
Figure 4.14: Configurations with 8 and 12 MCs.

Figure 4.15: Sensitivity to the number of MCs.

Figure 4.16: Sensitivity to the core count.

The main observation from these results is that our approach generates higher savings with larger memory controller counts. This is mainly because a larger number of controllers lead to better memory parallelism within each cluster, which in turn increases the effectiveness of our approach (as we are not hurt by a degradation in memory level parallelism when off-chip accesses are localized).

Results with Different Core Counts. We next present the sensitivity analysis to the core count. In Figure 4.16, we present results with a $4 \times 4$ multicore system with four controllers (one corner each, with the L2-to-MC mapping similar to Figure 4.4a). The results with our default configuration ($8 \times 8$) are reproduced
Figure 4.17: Application performance in the case that cache-block interleaving is adopted for SNUCA.

here for ease of comparison. We see an average improvement of 14%, meaning that our approach is successful with this configuration as well.

Results with Shared L2. Figure 4.17 shows the results collected when the available L2 space is managed as a shared SNUCA cache (with cache-block interleaving for both L2 and main memory). In general, the results shown here are similar to those given earlier in the case of private L2s (Figure 4.10). We observe that our approach (see Section 4.4.3.2) achieves better improvements with shared cache (over private L2 case), except in three benchmarks (art, raytrace, and filter-2). The average execution time improvement in the shared L2 case is about 24.3%. Overall, these results along with those presented earlier show that our approach works very well under both private and shared last level caches.
Chapter 5

Row-buffer Locality Optimization

5.1 Background

5.1.1 Cache Basics

Multiple levels of on-chip caches are common today in both single-core and multi-core machines. Most on-chip caches are organized as a collection of cache blocks (lines). A cache block is the unit of data transfer to/from an underlying layer in the memory hierarchy (DRAM or a lower level cache memory). Typical block sizes can vary between 32 bytes and 256 bytes. The mapping between memory blocks (a contiguous memory space of cache block size) and cache blocks is an important design issue. In a $k$-way set-associative cache, a memory block can be mapped to any of $k$ different cache blocks. The set (within all of the ways) is determined by the index in the address (which is divided into tag, index, block-offset and byte-offset bits). The number of sets in a way is $2^{\#\text{of index bits}}$. The tag is then compared to the tag field in the cache. If there is no index field (i.e., after block and byte offset, all the rest bits are tag), the cache is fully associative. If there is only one way, the cache is direct mapped. A computation is said to exhibit “data locality” if it reuses the data in a cache block before the line is displaced from the cache.

The memory layout (that determines which blocks are stored contiguously in memory) has a huge bearing on the cache behavior. Specifically, if the memory layout and data access pattern are in synch, then the limited capacity of the cache and associativity can be better utilized. As an example, let us assume that we
have a direct-mapped cache and two scalar variables \((a\) and \(b)\) are being accessed by a program in the order “\(a, b, a, b, \cdots\)”. If these two variables are laid out in memory such that they belong to two different memory blocks that map to two different cache blocks, then this access pattern leads to only two misses. On the other hand, if the memory layout is such that \(a\) and \(b\) map to two memory blocks that in turn get mapped to the same cache block, the same access pattern with the same cache organization may lead to all eight accesses being a miss (resulting in a cache miss rate of 100%). Therefore, layout of data in memory can play a critical role as far as cache behavior is concerned.

### 5.1.2 Row-buffer Basics

A row-buffer is a small buffer employed in modern memory chips that allows quick and easy access to multiple data words (usually the page size) located on a common physical row in the memory. In the rest of this paper, we use the terms “row” and “page” interchangeably. In a multi-bank memory system, each bank has a separate row-buffer. Row-buffer conflicts happen when concurrent memory accesses to different pages fall in the same bank. In this case, a different row is requested and the content of the row-buffer needs to be changed. Such a memory access is termed as a “row-buffer miss”. In contrast, if the data content of a particular row is in the row-buffer, subsequent memory requests to different columns in the same row can be serviced quickly from the row-buffer without having to access the memory bank or memory cell array. Such a memory access is termed as “row-buffer hit”.

The latency difference between a row-buffer miss versus row-buffer hit can be significant. In the former case, precharge and row access are needed to initiate each access and these operations cannot be pipelined. As a result, the data access patterns that belong to this category can experience much higher latencies compared to the case of a row-buffer hit, and only partially utilize the memory bandwidth. However, in the latter case, only the peripheral circuitry is utilized, and no access is necessary to the underlying memory cell array.

Row-buffer locality refers to the reuse of the same row while it is buffered.\(^1\) The memory layout of data can also affect the row-buffer locality. For example, let us

\(^1\)In this paper, we assume the open page policy [51] in the row buffer.
5.2 Motivation and Overview

Based on the discussion in Section 5.1, clearly, the overall performance of an application is not dictated only by the data locality in caches, but is also influenced by the row-buffer locality. Specifically, even if two different versions of a given application have the same number of cache misses, their execution time can be very different, mainly because not all cache misses would experience the same miss latency due to the row-buffer. Figure 5.1 illustrates a motivational example, where in Layout 1, the data elements that incur misses in the last level cache are distributed over four pages. From a row-buffer locality perspective, the accesses to these data elements are expected, in general, to cause more row-buffer misses compared to the case in Layout 2, where these data elements reside in two pages. As a result, although the accesses to the data elements in Layout 1 and Layout 2 incur the same number of cache misses, Layout 2 exhibits better row-buffer locality, and is therefore expected to achieve much better memory performance.

Based on this example, we want to reorganize a given (input) memory layout such that: (i) the number of misses in each layer of on-chip caches (e.g., L1, L2 and L3) will not be increased; and (ii) the number of row-buffer misses will be reduced. Note that, the input memory layout could be the original memory layout or a layout obtained from prior data reuse optimiza-
tions that target minimizing cache misses. In other words, our proposed approach is orthogonal to the existing cache-centric data locality optimizations. In fact, to maximize performance improvement, both cache locality and row-buffer locality optimizing techniques should be used together. As shown in Figure 8.3, our proposed scheme consists of two steps, namely, row-buffer locality test and row-buffer locality enhancement. In the first step, we test whether the target application is suitable for the row-buffer locality improvement strategy proposed in the second step. If this test fails, our framework terminates the optimization process and return. In most cases, this happens when the original memory layout already exhibits good row-buffer locality. However, if the test passes, we proceed to the second step which determines a suitable memory layout that satisfy constraints (i) and (ii) simultaneously. In this step, we employ a cache miss model which can be considered as the theoretical support to ensure that our proposed layout optimization scheme will not increase the number of cache misses across the cache hierarchy. One of the observations from our cache miss model is that, to satisfy constraint (i), reorganizing the layout should be performed at the granularity of memory blocks. Therefore, our second step actually consists of two substeps: identifying memory blocks and reshuffling memory blocks.

Before closing our discussion in this section, there are two issues we want to clarify. First, our proposed strategy does not need to distinguish the data accessed by different threads. Instead, it places the data blocks accessed in a given time period into the memory such that a “minimum number of row buffers” are occupied; the accesses to these data blocks can be from the same thread or different threads. Second, because of the generality of the problem we are trying to solve, our framework can be applied to any system with caches and row-buffers. However, depending on the system that the target applications are running on, changing the memory layout may also have impact on other factors that are critical to the overall performance. Therefore, we may need slightly different data placement strategies for different systems (more discussion will be presented in Section 5.4).
5.3 Row-buffer Locality-Aware Data Placement

In this section, we first discuss how to find a memory layout that exhibits good row-buffer locality in a system with only one row-buffer. We assume that the given loop nest contains only one array, and for now, we do not care whether the target application is single-threaded or multi-threaded (this will be dealt with in Section 5.3.3.1). This is the simplest case that can be used to explain our basic strategy. In Section 5.4, we explain how to extend this to the case of a multi-threaded application (with multiple arrays in a loop nest) running on a system with multiple row-buffers.

5.3.1 Program Representation

Our work targets loop and array intensive applications with affine loop nests, in which, array subscript expressions and loop bounds are affine functions of enclosing loop indices and loop-independent variables. Specifically, in these applications, a loop nest that contains $l$ loops represents an $l$-dimensional iteration space, and each point in this space (i.e., an iteration) can be denoted by an iteration vector $\vec{i} = (i_0, i_1, \cdots, i_{l-2}, i_{l-1})^T$, where $i_j (0 \leq j \leq l)$ can take values from $[L_j, U_j]$, $L_j$ and $U_j$ being, respectively, the lower and upper bounds of the $j$-th loop from the top. Similarly, the data space of an $n$-dimensional array can be viewed as an $n$-dimensional rectilinear polyhedron [45] bounded by the array bounds. Each data element (each point in this polyhedron) can be expressed by a data vector $\vec{a} = (a_1, a_2, \cdots, a_n)^T$, where $a_1, a_2, \cdots, a_n$ are the array indices. The mapping between iteration space and data space is represented by array references, each of
Figure 5.3: (a) and (b) illustrate two different data footprints on the data space of array $A$, as well as the corresponding code fragments. The black dots represent the data elements, the solid black arrows indicate the direction of the data footprint, and the dashed boxes represent the memory blocks accessed by successive loop iterations.

which can be written as $\vec{r} = A\vec{i} + \vec{o}$, where $A$ is an $n \times m$ constant matrix called the access matrix, and $\vec{o}$ is an $n \times 1$ constant vector. For example, the reference $A[i_1+1][2i_2-4]$ in a two-loop nest can be expressed as: $\vec{a} = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \cdot \vec{i} + \begin{pmatrix} 1 \\ -4 \end{pmatrix}$, where $\vec{i} = (i_1, i_2)^T$ and $\vec{a} = (a_1, a_2)^T$.

### 5.3.2 Row-buffer Locality Test

The row-buffer locality of an array is closely related to the data footprint of each reference to that array. A data footprint is the sequence/order of data elements that are accessed over the loop iterations through an array reference, and each “array reference” corresponds to a “data footprint”. Figure 5.3 illustrates two examples. In Figure 5.3a, the data footprint is only along the fastest-varying dimension. This means that the data (array) elements accessed by successive loop iterations are already stored in the same or nearby pages. Therefore, no matter which data elements incur cache misses, these data elements will occupy as few pages (memory rows) as possible, which is the case also depicted in Layout 2 in Figure 5.1. In other words, for the references that have this kind of data footprint, there is no need to perform any row-buffer locality enhancement. In contrast, the
data footprint in Figure 5.3b is orthogonal to the fastest-varying dimension, which in a sense represents the worst scenario as far as row-buffer locality is concerned, i.e., the data elements accessed by successive loop iterations are not expected to be in the same or nearby pages. Therefore, as long as we place the data elements accessed by successive iterations into consecutive memory locations, the row-buffer locality can be improved.

The basic idea of this step is to test if the data footprint of a target array is the fastest-varying dimension (i.e., the dimension whose index changes quicker than any other index as we go over array elements sequentially). If it is, then the default layout of the target array already exhibits “good” row-buffer locality. Therefore, there is no need to perform any row-buffer optimization. Otherwise, we proceed to the next step, where we try to improve row-buffer locality through data layout optimizations.

To determine the data footprint of the target array, we employ a concept called the direction vector, which is determined by individually considering each reference to the target array. This vector is obtained by tracing the data elements accessed by successive loop iterations (through the same array reference) in the innermost loop. Specifically, let us assume that \( \vec{i}_1 \) and \( \vec{i}_2 \) are two iterations that satisfy \( \vec{i}_2 = \vec{i}_1 + (0, \cdots, 0, 1)^T \). That is, \( \vec{i}_2 \) is the iteration immediately follows \( \vec{i}_1 \). Given an array reference \( \vec{r} = A\vec{i} + \vec{o} \), the two data elements accessed by these two iterations can be expressed as \( \vec{r}_1 = A\vec{i}_1 + \vec{o} \) and \( \vec{r}_2 = A\vec{i}_2 + \vec{o} \), respectively. Therefore, the difference between these two data vectors \( \vec{r}_1 - \vec{r}_2 \) can be expressed as \( A(0, \cdots, 0, 1)^T \), which is the last column of matrix \( A \), i.e., the direction vector. Note that, the direction vector can be used to represent the direction of the data footprint for each array reference. For example, if the value of the direction vector is in the form of \( (0, \cdots, 0, x)^T \), where \( x \) stands for any expression or constant, the data footprint is along the fastest-varying dimension. Therefore, in our row-buffer locality test, we simply test if all the data footprints for the target array are of this form.

At this point, there is an important issue that needs to be clarified. When the target loop nest contains multiple references to the same array, we only consider the footprint of the data elements accessed through each array reference individually, and do not consider the data access sequence among different references. This
is because the direction of this inter-reference data access sequence is not very relevant as far as the row-buffer locality is concerned. For example, let us assume that the statement $A[j][i] = A[j+1][i] + 1$ in Figure 5.3b is replaced with the statement $A[j][i] = A[j][i+1] + 1$. Then the direction of the inter-reference data access sequence is along the fastest-varying dimension, but for any given value of $i$ and $j$, if there exists row-buffer locality between $A[j][i]$ and $A[j][i+1]$, then these two data elements reside in the same memory block. This indicates that there is still an opportunity to improve the row-buffer locality.

### 5.3.3 Row-Buffer Locality Enhancement

After our row-buffer locality test, our next step places the data elements accessed by successive loop iterations into as fewer pages (memory rows) as possible (as illustrated in Figure 5.1) without increasing the cache misses. This requires us to first understand, under what conditions, or using what kind of data layout reorganizations, the cache misses would not be increased. Our cache miss model introduced below provides the theoretical support to achieve this.

#### 5.3.3.1 Cache Miss Model

To help abstract this problem, we introduce the following notation. We use $T$ to represent the data access footprint/sequence of a target array $A$ (which is indicated by the execution order of the loop nest that encloses the references to this array), and $a_x$ to represent the access to a particular data element $x$ (of array $A$) in $T$. Since the data elements are accessed sequentially in $T$, each access $a_x$ can be assigned a “logical time stamp”, which is denoted using $t(a_x)$. In addition, since each data element is placed into a memory block, each access to a data element in $T$ corresponds to an access to the memory block that contains that data element. We use $b_x$ to denote the memory block where $x$ resides and $a^{-}_x$ to represent the “most recent access” to $b_x$ before $a_x$, and $C(b_x)$ to denote the set of memory blocks that can be mapped to the same set of cache blocks in a $k$-way set-associative cache as $b_x$.

Next, we introduce the concept of block distance. The block distance between two accesses $a_x$ and $a_y$, where $t(a_y) < t(a_x)$, is the number of “distinct” memory
blocks that can be mapped to the same set of cache blocks as $x$ and accessed during time period $[t(a_y), t(a_x) - 1]$. In particular, when $t(a_y) = t(a_x)$, $dis(a_y, a_x)$ is referred to as the block reuse distance of $a_x$. If $a_x$ is the first access (or $a_y$ is the last access) to the memory block $b_x$ in $T$, we define $dis(a_y, a_x)$ as $\infty$. An example that illustrates the block reuse distance is given in Figure 5.4.

The block distance concept can be used to explain the cause of cache misses: given a $k$-way set associative cache and two accesses $a_x$ and $a_y$, where $a_y$ is the most recent access to $b_x$ before $a_x$, if $dis(a_y, a_x) \leq k$, then $a_y$ ensures that $b_x$ is in the target cache at time $t(a_x)$, and consequently, $a_x$ will incur a hit; otherwise, $a_x$ will incur a miss. As can be observed, whether an access to a data element will generate a cache miss or hit depends only on the cache block where that data element resides, and not on the exact position within the cache block. Further, whether an access to a data element will generate a row-buffer miss or hit depends only on the page (row) where this data element resides, and is independent of the position within the page. Based on this observation, our memory layout transformation is performed at a cache block granularity, and we have the following lemma:

**Lemma 5.3.1.** Moving all the data elements in $b_x$ to $b_y$ will not change the block distance of any access in $T$ if $C(b_y) = C(b_x)$.

**Proof.** First, for any access $a_z$ that originally incurs a hit in the target cache, if
$C(b_z) \neq C(b_x)$ or there is no access to the data elements in $b_x$ within time period $[t(a^-_z), t(a_z)]$, moving all the data elements in $b_x$ to $b_y$ will not change the block distance of $a_z$. Second, in cases where we have $C(b_z) = C(b_x)$ and there exist accesses to the data elements in $b_x$ within time period $[t(a^-_z), t(a_z)]$, after moving all the data elements in $b_x$ to $b_y$, none of the distinct memory blocks in $C(b_x)$ that are accessed during time period $[t(a^-_z), t(a_z) - 1]$ will change except that $b_x$ will be replaced by $b_y$. As a result, in this case, the block distance of $a_z$ will still remain the same.

In conclusion, if we divide the data elements into memory blocks, and move these memory blocks in the memory space as suggested by Lemma 5.3.1, the block distance of each access to these data elements will not be changed, and therefore, the total number of cache misses will not be changed. Further, in most cache hierarchies, the cache block size at the lower level (e.g., L2) is the multiple of the cache block size at the higher level (e.g., L1). As a result, we reshuffle the memory blocks, whose size is the same as the last-level cache block size, the total number of cache misses will not be increased at any cache level.

It is important to note that, Lemma 5.3.1 is irrespective of whether the data elements in $b_x$ are accessed by multiple threads (as in the case of a multi-threaded application) or a single thread, and irrespective of whether these data elements belong to different arrays or to a single array, as long as $b_x$ and $b_y$ are mapped to the same set of caches. That is to say, this lemma is independent of how the data access sequence is formed (it could be formed by multi-threaded application with multiple arrays being accessed). We will later use this conclusion in Section 5.4.

### 5.3.3.2 Identifying Memory Blocks

To identify the memory blocks with the same size of the last-level cache block, we first employ padding [3] to align data elements within an array to make the fastest-varying dimension divisible by $s \times c$, where $s$ is the memory block size in terms of the number of data elements and $c$ is the number of cache sets in a $k$-way set associative cache. Specifically, let us assume that the size of the original array is $n_1 \times n_2 \times \cdots \times n_m$. If the value of $n_m$ is not divisible by $s \times c$, then we increase the size of the array along the $m$-th dimension from $n_m$ to $\lfloor n_m/(s \times c) \rfloor \times (s \times c)$,
and the additional entries in the array are left empty. The reason behind this is to ensure that, after strip-mining (explained shortly), the data elements that have the same array index along a certain dimension can be mapped to the same set of cache blocks. This will help us easily identify which memory position (location) that a given memory block should be reshuffled to in the second step (based on Lemma 5.3.1). Note however that, since padding increases the size of the array, it changes the default memory layout. Therefore, it could also have a negative impact on cache performance, i.e., the number of cache misses could potentially be increased. To solve this problem, we also employ a compiler-based cache miss prediction scheme built upon CME (cache miss equations [52]) to predict the total number of misses before and after padding. If the predicted number of cache misses after padding is increased, we stop our optimization process and return. Otherwise, we continue as follows.

We employ strip-mining [46] to divide the data space (array space) into memory blocks. Let \( n_m \) denote the size of the array along the fastest-varying dimension. Then, strip-mining this dimension with a memory block size of \( s \) transforms it to two dimensions with the size of \( n_m/s \) and \( s \); and a reference \( \vec{r} = (\cdots, r_m) \) becomes \( (\cdots, r_m/s, r_m \% s) \). After this transformation, \( r_m/s \) and \( r_m \% s \) can be used to identify the index of the block and the offset within a block, respectively.

Note that, although this transformation changes the array indices, it does not change the layout of the array. Actually, it just provides another view for the array layout that helps us identify the memory blocks to which each data element belongs. Therefore, it does not change the number of cache misses incurred during the execution of the target application. In our current implementation, we employ the techniques suggested in [46] to reduce the potential runtime overheads of these transformations.

### 5.3.3.3 Reshuffling Memory Blocks

We next focus on reshuffling these memory blocks in the memory space to improve row-buffer locality. We employ a concept called best candidate dimension (BCD) for the target array, which is the dimension along which data elements need to be laid out to maximize row-buffer performance. For example, in Figure 5.3a and Figure 5.3b, the BCDs are the fastest-varying dimension and the second fastest-
varying dimension, respectively. Note that BCD can be determined by the direction vectors. This is because the value of each entry of a direction vector can be used to evaluate the row-buffer locality along the corresponding dimension. Let $d_k$ denote the value of the entry on the $k$-th dimension of the direction vector. Then, larger the value of $d_k$ is, better row-buffer locality the array exhibits along the $k$-th dimension. For example, the direction vector $(1, 2)^T$ indicates that the row-buffer locality along the fast-varying dimension (the second dimension) is better than the one along the first dimension. Specifically, let us assume that there are $n$ direction vectors for an $m$-dimensional array, which can be represented as $(d_{x,1}, d_{x,2}, \cdots, d_{x,m})^T$, where $1 \leq x \leq n$. We calculate the sum of the entries in each dimension for these direction vectors as $d_1 = \sum_{x=1}^{n} d_{x,1}$, $d_2 = \sum_{x=1}^{n} d_{x,2}$, $\cdots$, $d_m = \sum_{x=1}^{n} d_{x,m}$ and select the dimension that has the largest sum as the BCD.

Figure 5.5 shows the basic strategy adopted in this step for the example in Figure 5.3b. Intuitively, the BCD is orthogonal to the fastest-varying dimension. Since in most array/loop intensive applications, the array bounds are very large, the data elements that are accessed by the innermost loops can easy span multi-
ple pages. In comparison, if we place the memory blocks accessed by successive iterations into consecutive memory locations (as indicated by the arrows), the number of pages occupied by these memory blocks is reduced, and consequently, the row-buffer locality can be improved.

We use Lemma 5.3.1 to ensure that the number of cache misses at each cache level will not be increased as a result of this transformation. Specifically, let us consider the optimization strategy shown in Figure 5.6, which maintains the cache miss rate in the last level cache. Observe that, after our first step (identifying memory blocks), we know the last level cache set to which a given memory block is mapped. For example, in this figure, the memory blocks in the first column will be mapped to set 0, and the memory blocks in the second column will be mapped to set 1, and so on. Therefore, the data elements in memory blocks $b_1$, $b_2$, $b_3$, and $b_4$ will be mapped to the same set of cache blocks. Clearly, according to Lemma 5.3.1, placing the data elements in memory block $b_1$ (or $b_2$) into memory block $b_4$ (or $b_3$) will not change the number of cache misses at the last-level cache.

Based on this discussion, our memory block reshuffling strategy can be explained as follows. First, to improve the row-buffer locality, we place the memory blocks (corresponding to the last level cache block) along the BCD into the memory positions along the fastest-varying dimension. Second, to retain the original number of cache misses at each cache level, we need to maintain a distance between these new memory locations (in terms of the number of memory blocks) such that the set of cache blocks that each data element is mapped to (at each cache level) will not be changed. To obtain this layout, we modify the array indices obtained using the strategy discussed in Section 5.3.3.2, which is in the form of $(\cdots, r_x, r_m/s, r_m\%s)^T$, as follows. Let us assume that the $x$-th dimension is the BCD (note that the value of $r_x$ represents the $r_x$-th memory block along this dimension). To make our explanation easy to follow, in the rest of our discussion, we only give the transformed array index form for the case of $x = 2$. To maintain the last level cache miss rate, we move the data elements in this memory block to the $(r_m/s + r_x \times c + 1)$-th memory block along the fastest-varying dimension. In other words, we keep the distance between two memory blocks along the $x$-th dimension as $c$ (see Figure 5.6). Therefore, the array indices after shuffling the memory blocks can be expressed as $(\cdots, r_m/s + r_x \times c + 1, r_m\%s)$. Similarly, to
Algorithm 3 Determining the row-buffer locality-aware memory layout for each array.

INPUT: Array references $\vec{r}_1, \vec{r}_2, \cdots, \vec{r}_n$;

OUTPUT: Transformed array references $\vec{r}'_1, \vec{r}'_2, \cdots, \vec{r}'_n$;

1: Calculate the direction vector for each array reference, which is in the form of $(d_{x,1}, d_{x,2}, \cdots, d_{x,m})^T$, where $1 \leq x \leq n$;
2: Calculate the sum of entries on the each dimension for these direction vectors as $d_1 = \sum_{x=1}^n d_{x,1}, d_2 = \sum_{x=1}^n d_{x,2}, \cdots, d_m = \sum_{x=1}^n d_{x,m}$;
3: Select the BCD as $d_x$;
4: if $x == n$ then
5: Return;
6: end if
7: /*Row-buffer locality enhancement */
8: /*Identifying memory blocks */
9: Use CME ([52]) to predict the row-buffer misses after padding;
10: if the cache miss rate at any cache level increases then
11: Return;
12: else
13: Identify the memory blocks by transforming the original reference $\vec{r}_k = (\cdots, r_x, \cdots, r_m)^T$ to $(\cdots, r_x, \cdots, r_m/s, r_m \% s)^T$;
14: end if
15: /*Reshuffling memory blocks */
16: Reshuffle the memory blocks by transforming the above reference to $(\cdots, r_m/s + r_x \times c + 1, r_m \% s)^T$;
17: return The transformed array reference.

Maintain the cache misses rate at a higher level cache as well, we only need to change the distance mentioned above to $lcm(c, c')$, where $c'$ is the cache block size at the higher level. Note however that, since $c$ is always the multiple of $c'$, we have $lcm(c, c') = c$. This means that $(\cdots, r_m/s + r_x \times c + 1, r_m \% s)$ actually maintains the cache miss rate (of the input layout) at each cache layer in the hierarchy. As a result, this is the final format of the transformed array reference. Our data layout transformation algorithm is given as Algorithm 3.

5.4 Multiple Arrays and Multiple Row-Buffers

In this section, we extend the layout optimization strategy discussed so far to the case where the input code contains multiple arrays that are manipulated in an
NoC-based multicore system (network-on-chip [1]) with multiple row-buffers (our approach is not tied to NoC; it can be used with other types of CMPs as well). Figure 5.7 shows an example NoC-based multicore with 16 cores and 16 row-buffers, where each core has a shared L2 (cache) bank, and each bank corresponds to a row-buffer.

In this multiple-row-buffer and multiple-array case, we have two additional issues to address. First, if the target loop nest contains references to multiple arrays, the row-buffer locality will be destroyed by the interleaving data accesses to different arrays. For example, let us assume that the target loop nest has a statement of the form $A[i][j] = B[i][j] + C[i][j]$. Since we have three array accesses within the same iteration, the accessed data elements cannot reside in the same page (as-
Figure 5.9: The left portion is the original code and the right portion is the code after generating the initialization loop. Note that 256 in this loop is the page (row) size in terms of the number of data elements.

Figure 5.10: Reshuffling the memory blocks for the case of multiple row-buffers. Now, instead of keeping a distance of $c$ (the case of single row-buffer), we keep the distance as $lcm(p, c)$.

assuming that arrays are large enough. Therefore, when the pages touched by these accesses are mapped to the same memory bank, row-buffer conflicts will happen, which will in turn offset any benefit coming from optimizing row-buffer locality of each array in isolation. The solution to this problem is to ensure that the pages touched by the accesses in the same iteration are always mapped to different memory banks. Depending on the page allocation policy adopted by the OS, there are two ways to achieve this. First, if the page coloring policy is adopted, consecutive virtual pages are mapped to consecutive colors. Therefore, one can modify the default page coloring algorithm in the OS to ensure the above mapping [53]. Second,
if the bin-hopping policy is adopted, a physical page is allocated to a virtual page only when this virtual page is accessed at the first time. Therefore, we can add an initialization loop nest right before the target loop nest to make the physical pages touched by the accesses to different arrays in each iteration consecutive (therefore not overlapped). An example is shown in Figure 5.9.

The second challenge is related to the data-to-L2 bank mapping scheme employed by the underlying hardware. Figure 5.8 shows the mapping of a physical memory address to the L2 (last level) cache banks of an NoC based multicore. The address space is block-cyclically mapped across the banks. More specifically, the lowest bits represent the offset within a memory block mapped to a bank, and the next set of bits specify the bank number. This data-to-L2 bank mapping scheme brings an additional challenge: when applying the memory block reorganization presented in Section 5.3.3.3, we also need to ensure that the L2 bank that each memory block is mapped to will not be changed. The reason is that, in an NoC-based multicore with shared L2 caches, a data request from a node will first be forwarded to the home node of this data element, which is co-located with the L2 bank to which this data is mapped. If we do not have this requirement, the on-chip data access latency might change. Therefore, compared to the original case, an on-chip data request from a node may experience a longer latency on the network to reach the home node of this data (the home node is far away from the request node in this case), which can in turn degrade application performance. Further, if we do not take the impact of memory block reshuffling on the on-chip data access latency into account, it is hard to realize the potential performance improvements of our row-buffer locality optimization. To address this problem, we slightly modify our memory block reshuffling strategy discussed in Section 5.3.3.3. Specifically, assuming that the number of L2 cache banks is \( p \) and the number of cache sets is \( c \), we want to keep every two memory blocks that can be mapped to the same set of caches at a distance (in terms of the number of memory blocks) of \( \text{lcm}(p, c) \), which is the least common multiple number of \( p \) and \( c \), along the fastest-varying dimen-

---

2The granularity of interleaving across the banks can range from one cache block to one or more memory rows. However, cache block interleaving has a number of benefits, including uniform distribution of arrays across the banks, enabling high collective bandwidth in accessing data elements, and minimization of hot-spots [1]. In this case, the bits that determine the data-to-L2 bank mapping in the physical address are the same as the corresponding bits in the virtual address. Because of these reasons, the evaluations in this section assume cache-line interleaving.
Table 5.1: Experimental setup.

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>L3 Miss Rate (%)</th>
<th>Row-Buffer Miss Rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>wupwise</td>
<td>Quantum chromodynamics</td>
<td>16.3</td>
<td>59.3</td>
</tr>
<tr>
<td>swim</td>
<td>Shallow water modeling</td>
<td>21.2</td>
<td>71.2</td>
</tr>
<tr>
<td>mgrid</td>
<td>Multi-grid solver in 3D potential field</td>
<td>24.3</td>
<td>62.2</td>
</tr>
<tr>
<td>applu</td>
<td>Parabolic/elliptic partial differential equations</td>
<td>18.1</td>
<td>68.7</td>
</tr>
<tr>
<td>galgel</td>
<td>Fluid dynamics</td>
<td>9.9</td>
<td>57.3</td>
</tr>
<tr>
<td>apsi</td>
<td>Solves problems regarding temperature, wind, and velocity</td>
<td>19.3</td>
<td>36.6</td>
</tr>
<tr>
<td>gafort</td>
<td>Genetic algorithm</td>
<td>16.9</td>
<td>68.2</td>
</tr>
<tr>
<td>fma3d</td>
<td>Finite element crash simulation</td>
<td>12.4</td>
<td>69.4</td>
</tr>
<tr>
<td>art</td>
<td>Neural network simulation</td>
<td>17.6</td>
<td>39.0</td>
</tr>
<tr>
<td>ammp</td>
<td>Computational chemistry</td>
<td>14.6</td>
<td>57.1</td>
</tr>
<tr>
<td>gauve2</td>
<td>Locates the types and formation of naval vessels in ocean</td>
<td>28.5</td>
<td>71.1</td>
</tr>
<tr>
<td>hyper</td>
<td>Hypergraph partitioning</td>
<td>22.0</td>
<td>78.3</td>
</tr>
</tbody>
</table>

Table 5.2: Benchmarks.

sion (see Figure 5.10). Similar to the analysis in Section 5.3.3.3, the array indices after reshuffling the memory blocks can be expressed as \( (\cdots, r/s + d \times \text{lcm}(p, c) + 1, r\%s) \).

## 5.5 Experimental Evaluation

### 5.5.1 Setup and Applications

This section presents an experimental evaluation of our proposed row-buffer optimization scheme. Table 5.1 gives the important parameters of the simulated multicore system. This system (which is similar to the one shown in Figure 5.7 as far as node architecture is concerned) consists of 32 cores connected as a \( 4 \times 8 \) grid through an on-chip network, and has 4 memory controllers (one on each corner). Each memory controller employs an open row policy and the default memory scheduling policy in FR-FCFS, where accesses in the memory controller that target the current row (page) are prioritized over others. And, if no reuse of the row-buffer
contents is possible (i.e., all requests queued in the memory controller target rows other than the one currently in the row-buffer), we simply use first-come, first-served. Each node in this architecture has private L1 and L2 caches. On the other hand, the shared L3 cache (last level) is distributed across all nodes. For each application program we have, we simulated 2 billion instructions, after fast-forwarding the initialization code. The set of multithreaded application programs used in this study is given in Table 5.2. Our programs include the benchmarks in SpecOMP [28] as well as two large codes: gsurvey2 and hyper. The first of these implements a defense surveillance algorithm, which classifies objects detected in an area of ocean depending on their length, breadth, area, and perimeter, whereas the second one performs hypergraph partitioning based on the checkerboard algorithm. For the SpecOMP applications, we used the standard inputs; the total data manipulated by two large applications are 870MB and 1.23GB. The third column of Table 5.2 gives the last-level (L3) cache miss rates and the fourth column shows the row-buffer miss rates. We see that, as far as row-buffer locality is concerned, our programs exhibit quite diverse performance, helping us cover a large spectrum.

We implemented our approach using the Open64 compiler as a source-to-source translator [54], and both the original and optimized programs are then compiled using gcc 4.7.2. In all the results presented below, the input codes to our approach have already been optimized (using loop permutation) for the last-level cache locality. And, our savings are given with respect to these cache locality-optimized codes.

### 5.5.2 Results

The first bar for each program in Figure 5.11 shows the percentage reduction in row-buffer miss rates as a result of our approach. We see that the improvement in row-buffer misses ranges between 19% and 56.4% in SpecOMP applications, averaging on 34.2%. The improvements with our two big applications are higher (59.6% and 45.2%). The second column in the same plot gives the percentage reduction on average memory (data) access latency. It can be seen that our approach achieves an average of 29% improvement in data access latency. Finally, the last column gives the reduction in execution cycles, which range between 5.9% and 22.1%,
averaging on 15.1%. Overall, these results indicate that our proposed approach is very effective in improving performance of these application programs that have already been optimized for cache locality.

We next compare our approach to two alternate optimization strategies. The
first of these is a single core (single thread) centric optimization strategy where accesses issued by each core are reorganized to maximize row-buffer hits. This approach in a sense represents the strategy proposed by a prior work [23]. The second approach is hierarchical tiling, as proposed by [55]. In this approach, the row-buffer is treated as an extra layer in the memory hierarchy and taken into account in determine tile shapes and sizes. We see from the results plotted in Figure 5.12 that our proposed approach generates much better results than both these approaches except in two programs. First, with respect to the approach in [23], our approach finds more opportunities as we employ layout optimization not loop restructuring (which is constrained by intrinsic data dependences in the code). Further, most of these application programs are parallelized such that data sharing among threads is minimized, i.e., they do not share data but they share pages (rows). A single core centric strategy does not exploit inter-thread page sharing and this results in a significant drop in potential improvements. Similarly, the hierarchical tiling approach [55] has a limited scope for optimization as any intervening access (i.e., an access that falls outside the current row) triggers a replacement of the entire row, and this prevents tiling to take advantage of row-buffers fully. The average execution cycle improvements brought by our approach, single-core centric approach and hierarchical tiling are 15.1%, 8.1% and 5.9%, respectively. In apsi and art, the results obtained through different schemes are very similar as they result in the same off-chip data access pattern.

To our knowledge, it is not possible to obtain row-buffer statistics from current commercial systems. However, we also tested our approach on an Intel Xeon 3500 series multicore system. This system includes 4 cores and a cumulative last level cache space of 8 MB. When our approach is applied, we observed that there was no major change in the last level cache misses (compared to the baseline, which is a data locality-optimized code as stated earlier), yet the execution times of our benchmark codes reduced between 4.4% (mgrid) and 13.8% (gafort), averaging on 9.8%. Since the only difference between the baseline codes and our optimized versions is the data layout optimization we performed, we can attribute these execution time improvements to our layout optimization.
5.5.3 Summary of the Sensitivity Experiments

In each sensitivity experiment presented below, the value of only one parameter is changed; values of all other parameters retain their original values given in Table 5.1. In Figure 5.13, every three bars form a group. Each group corresponds to the result from one configuration. In each group, the first-bar shows the average percentage reduction (under that configuration) in execution-cycles for SpecOMP-benchmarks. The second and third bars correspond to the reduction in execution-cycles for gsurve2 and hyper. Specifically, the second group of bars in Figure 5.13 corresponds to the improvements provided by our approach when the underlying memory scheduling policy is FCFS (recall that our default policy was FR-FCFS). We see that, although our savings reduce with this policy (as accesses to the same row are not rewarded as much), our approach still brings an average improvement of 17.6% when all applications are considered. The third group of bars gives the results with a smaller system (16 cores with 2 memory controllers). We see that average improvements in this case are quite similar to those obtained with the larger system. Next, we give the results under a smaller L3 capacity. We see that our savings significantly increase in this case as pressure on row-buffers increases. We next change the number of banks per memory controller (from 16 to 8), and observe that there is little change in our percentage improvements. Finally, we change the row (page) size from 8KB to 4KB, and observe increased improvements since a smaller row size makes locality a more critical concern.

\[^3\text{Since it is very hard for us to increase the dataset sizes of our programs (as it requires an in-depth understanding of the program), we hope to achieve a similar scenario by reducing the cache size.}\]
Related Work

6.1 Loop Transformation

Prior work on loop transformation and computation scheduling targeting both uniprocessors and multiprocessors has been done by many researchers [56, 35, 57, 58, 59, 34, 60, 61, 62, 63, 64, 32, 65, 66]. Prior code and data mapping strategies – developed in the context of traditional parallel systems – include [67, 68, 69, 70, 71]. Prior parallelization, scheduling, and mapping strategies outside the multicore domain also include [72, 73, 74, 75]. The main goal behind most of these studies is to exploit data-computation affinity, that is, to collocate computation and data carefully to improve locality of accesses. Our target architecture (multicores with shared cache hierarchies) and optimization goal (maximizing on-chip cache locality for shared data) are different from the targets (mostly distributed memory architectures with a couple of studies on SMPs) and goals (maximizing the number of local data accesses) of these prior studies. Consequently, our proposed algorithms are different from theirs.

Emergence of multicore architectures renewed interest in code parallelization [76, 77, 78, 79]. However, these parallelization strategies do not consider the target architecture explicitly. Sarkar and Tullsen propose a data-cache aware compilation to find a layout for data objects which minimizes inter-object conflict misses [80]. Zhang et al study reference affinity and present a heuristic model for data locality [81]. Zhang et al study the impact of cache sharing in multicores [82]. Kandemir et al [83] explore a code optimization strategy targeting only caches shared by all
cores. In comparison, our loop transformation strategy can work with any on-chip multi-layer cache hierarchy. Kandemir et al also propose a code mapping strategy based on polyhedral model [31]. In our experimental analysis, we compared our approach against [31]. Kurzak et al [84] present a scheduling strategy targeting dense linear algebra operations. In comparison, we propose a linear algebra based optimization framework that can handle both mapping and scheduling problem. In addition, the uniqueness of our approach is that we determine a cache topology-aware mapping based on reuse distances, rather than the amount of data shared as [83]. Our approach works better in practice, since even if two iterations share a large amount of data, their reuse distance may still be long, in which case, mapping them to the two cores that share a cache at a very high level (e.g., L1) may not help in exploiting the data reuse between them.

6.2 Data Transformation

There are also a large amount of work on data transformations. Chatterjee et al [85] present an automatic framework to determine array alignments in data-parallel languages such as HPF. Leung and Zahorjan [2] focus on array restructuring to improve the spatial locality based on an invertible linear transformation of array index vectors. Rivera et al [3] employ inter-variable padding and intra-variable padding to eliminate conflict misses. O’Boyle and Knijnenburg [4] describe the use of non-singular data layout transformations to improve program performance. Barua et al [86] introduce a compiler-managed memory system called MAPs that interleaves consecutive array elements in a round-robin manner across the memory banks of the RAW architecture. So et al [5] present an automatic framework that derives custom data layouts in multiple memory banks for array-based computations. Lu et al [87] develop a compile-time framework to optimize data locality for NUCA chip multiprocessors via non-canonical data layout transformation. Compared to our work, they focus on a NUCA type of architecture and their main goal is to reduce the number of non-local accesses. In contrast, our goal is to determine a customized memory layout based on the underlying multi-layer cache hierarchy of the target machine, with the goal of reducing the inter-thread conflicts in shared caches. In addition, there have been several efforts trying to integrate loop and
data transformations [88, 37, 41, 89, 90]. These efforts target uniprocessors. As compared to these studies, our work consider data accesses of multiple threads and determine a suitable layout for each target array based on the underlying cache topology.

6.3 Off-chip Data Localization

Most of the prior compiler-directed layout transformation are oriented towards improving cache performance. Leung and Zahorjan [2] propose array restructuring to improve the spatial locality in nested loops. O’Boyle and Knijnenburg [4] describe a unifying framework for non-singular data transformations. Both these works and our work employ the transformation matrix. However, while these prior works focus on improving data locality (cache performance), we focus on tracking the data elements touched by different threads. Further, since our desired transformation matrix $U$ is only characterized by its $v$-th row vector, one can further take the data locality into account by adding additional constraints on $U$. Therefore, our work is also orthogonal to these works.

Rivera and Tseng [3] introduce padding to eliminate conflict misses. Anderson et al [46] propose an integrated computation parallelization and data layout transformation framework, and use strip-mining and permutation to improve data locality. We employ these data transformation techniques to address different problems. In our context, padding is used to align the based address of inter-array and intra-array to ensure each data element has the desired virtual address in the transformed layout; and strip-mining and permutation are used to ensure that the off-chip accesses can be sent to the desired memory controller. Since our goal is different from these works, we generate different transformation formulas.

addresses the trade-off between cache access latency and number of off-chip accesses. Our page allocation algorithm are much simpler than these works since we take advantage of the data transformation to simplify the job for the OS. We also want to emphasize that, the page allocation scheme in our second component is only a complement to the first component.

Lu et al [12] develop a data layout transformation framework for enhancing NUCA-based chip multiprocessors by reducing non-local L2 accesses for localizable computations. There are two significant differences between our work and this work. First, they focus on on-chip accesses, while our goal is to investigate the impact of off-chip accesses on on-chip network. Second, we handle both shared L2 and private L2 caches while the previous works target only shared L2. Consequently, the theory we developed is entirely different from theirs.

There has been a large body of prior work addressing the problem of data distribution on large-scale NUMA systems [91, 92, 93]. Marathe et al [91] present a profile-based memory placement methodology. Navarro et al [92] discuss a compiler-based scheme to find the iteration and data decompositions that minimize communication and load imbalance overheads in parallel programs targeted at NUMA architectures. Majo and Gross [93] study the data distribution of a program to the individual and multiple data access patterns on NUCA. However, there is an important difference between our work and prior NUMA oriented work - our approach handles the case with shared on-chip caches, a unique characteristic of multicores. Recent architecture based works involve memory scheduling of off-chip accesses (e.g. [14, 16, 94]), which are orthogonal to our compiler based approach. That is, they can be used together with our approach.

### 6.4 Row-buffer Locality

There exist a large number of studies on data transformations. Leung and Zahorjan [2] focus on array restructuring to improve the spatial locality based on an invertible linear transformation of array index vectors. Lu et al [12] develop a compile-time framework to optimize data locality for NUCA chip multiprocessors via non-canonical memory layout transformation.

So et al [5] present an automatic framework that derives custom memory lay-
outs in multiple memory banks for array-based computations. Zhang et al [95] discuss a memory layout transformation that considers both data access patterns exhibited by application threads and the on-chip cache topology of the target multicore architecture, which minimizes cache conflicts across threads. Cierniak and Li [41] present a unified approach to locality optimization that employs both data and control transformations. Zhang et al study reference affinity and present a heuristic model for data locality [81]. Rivera et al [3] employ inter-variable padding and intra-variable padding to eliminate conflict misses. Anderson et al [46] develop a framework that automatically parallelizes sequential programs and changes the original array layouts to improve memory system performance. Our layout customization step employs the non-linear data transformation techniques proposed by them. However, these techniques are applied for different purposes in our context and therefore our problem formulation is entirely different.

Recent computer architecture research targets at improving memory system performance [14, 15, 16, 17, 18, 19, 20, 21, 22]. As mentioned before, prior compiler work [23] on optimizing for row-buffers targeted single-threaded embedded applications and employed loop transformations. The experimental comparison against it (in Section 5.5) indicates that our data layout optimization strategy generates much better performance improvements, primarily due to two factors. First, our approach is not constrained by data dependences in the target code, and second, it has a global view of the target application, i.e., it considers the data elements that are accessed by all threads (instead of trying to optimize the row-buffer performance of each thread/core in isolation, which may not be very effective as interleaving accesses from parallel threads can easily destroy row-buffer locality).
Conclusion

In this thesis, we first present two approaches that target improving the data locality by considering the underlying cache hierarchy. The goal of our first scheme is to maximize constructive sharing and minimize destructive interferences, in the context of multicores, through careful mapping and scheduling of loop iterations considering on-chip cache topology. We implemented our mapping and scheduling schemes by extending a compiler infrastructure and tested their impact using a commercial multicore machine and eight application programs. The experimental results collected show that our mapping scheme improves the L1, L2 and L3 cache miss rates by, respectively, 13.5%, 16.6% and 14.9% on average, resulting in an average execution time improvement of 14.1%. Our scheduling supports take these improvements to higher values.

In contrast, our second work proposes a novel data layout optimization strategy that targets multithreaded applications running on multicore platforms. The uniqueness of this strategy is that it determines a memory layout for an array, customized based on data access patterns of all threads as well as target on-chip cache topology. To measure the gains brought by this approach, we automated it within a compiler infrastructure and tested using 22 benchmark programs on a commercial multicore machine. Our experimental evaluation shows that this approach can generate significantly better results for most of the benchmarks when compared against different state-of-the-art code and layout optimization techniques.

Our third scheme targets optimizing “off-chip accesses” in both private L2s and shared L2 based multicores. Specifically, it proposes a compiler-guided off-
chip access locality optimization strategy with the goal of minimizing the distance between the requester core/node and the target memory controller. Doing so brings two benefits: first, off-chip data accesses are expedited as they travel shorter distances over the on-chip network, and second, on-chip data access latencies are also reduced as, after the optimization, they are less by affected by off-chip accesses. Our results with 12 multithreaded applications, in the case of private L2s, indicate an average of 18.2% saving in execution time. The corresponding execution time improvement, in the case of shared L2 is about 24.3%.

Our fourth scheme targets a compiler-directed row-buffer locality optimization. This strategy transforms memory layouts of data arrays with the goal of (i) increasing the number of row-buffer hits and (ii) not increasing the number of cache misses at any layer in the on-chip cache hierarchy. The results collected from our compiler implementation and experiments are very encouraging and indicate about 15% improvement in execution time when averaged over all multithreaded programs tested.

It should be noted that, in this thesis, all the experiments are performed on one application at a time. This is because, if we run multiple application together, there will be many combinations, which will make it harder to predict the overall application behaviors (e.g., data access pattern). This in turn will provide the compiler less opportunity of the proposed optimizations.
One of our future work is to shape the performance of last-level cache misses for “memory parallelism”. While there are different definitions of this term in the literature [96, 97, 98], the version considered in our future work is “bank-level parallelism” (BLP), which can be defined as the number of concurrent accesses to different memory banks in the system. BLP can change from one workload/application to another or even across different phases of the same workload/application and, as documented by the previous research [99, 100, 101], a high BLP usually correlates very well with low memory access latency. Figure 8.1(a) gives the bank-level parallelism (BLP) and cycles per load instruction (CPL) numbers for our original benchmarks (averaged over the entire execution), when they are tiled (i.e., iteration spaces of loops are divided into small chunks) and tiles are scheduled without considering BLP. One can observe from this plot that the resulting BLP values are not high, considering the fact that our default memory system has 64 banks. In addition, we see that a low BLP usually corresponds to a high CPL and vice versa. As a result, improving BLP can help us reduce the memory cycles, and eventually, improve overall application performance. Figures 8.1(b), (c) and (d) give the variations of BLP over the course of execution for three of our applications. One can identify from these plots the different execution phases (roughly, corresponding to different loop nests) from the perspective of BLP. For example, in gafort, we have a phase between epochs 24 and 36, and another phase between epochs 38 and 45. Our proposed approach in this work tries to optimize BLP in each major phase of an application.
While there are several papers [102, 100, 103] that propose hardware-based techniques that quantify BLP and/or optimize it, we are not aware of any compiler-based effort on optimizing BLP in multithreaded applications that targets multi-
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        X(i, j) = X(i, j-1) + X(i-1, j-1);
    }
}

for (ii = 0; ii < N; ii += 2) {
    for (jj = 0; jj < N; jj += 2) {
        for (iii = ii; iii < min(ii + 2, N); iii++) {
            for (jjj = jj; jjj < min(jj + 2, N); jjj++) {
                X(iii,jjj) = X(iii,jjj-1) + X(iii-1,jjj-1);
            }
        }
    }
}

(i) Before tiling.

(ii) After tiling.

Figure 8.2: (a) An example of loop tiling. (b) The corresponding iteration space. (c) The corresponding data dependence graph (DDG). Arrows capture the data dependences among tiles.

Figure 8.3: The place of our proposed scheme in the compilation flow.

core and manycore systems. Compared to hardware-based schemes, a compiler-based approach can have a limited scope (namely, compiler-analyzable codes).
However, in cases where it is applicable, it can bring much better improvements than hardware schemes as the compiler can analyze future data access patterns and restructure code to maximize BLP. Further, in many cases, a compiler-based scheme can also complement the hardware schemes.

Our proposed BLP-aware scheduling is based on a concept called *iteration space tiling* (also called loop blocking), which partitions a loop’s iteration space into small tiles or blocks. Figure 8.2(a) shows an example code before and after tiling, and Figure 8.2(b) illustrates the corresponding iteration space. There is a rich literature on tiling, focusing on both loop-level parallelism and data locality aspects [104, 105, 106, 107, 108, 109]. When tiles are used as the “units” of parallel execution, many existing compiler-directed loop optimization techniques [110, 111, 112, 108] make use of a *Data Dependence Graph* (DDG) (in some form) to guide the tile scheduling decisions. Figure 8.2(c) illustrates the DDG that corresponds to the tiled code shown in Figure 8.2(a). This graph can be constructed by the data-dependence analysis in the compiler and gives execution constraints among different tiles. More specifically, each node in a DDG represents a tile, and an edge from one node to another indicates a data dependence between corresponding tiles. For example, in Figure 8.2(c), tiles $T_4$, $T_5$, $T_6$ and $T_7$ cannot be executed until the execution of iterations in $T_0$, $T_1$, $T_2$ and $T_3$ is finished, due to the data dependence among them (captured by arrows). Therefore, $\{T_4, T_5, T_6, T_7\}$ and $\{T_0, T_1, T_2, T_3\}$ form two *dependence-free sets*, and in each set, the tiles can be scheduled at the same logical time slot (at different cores) to maximize application (loop) level parallelism. In most loop/data-intensive programs, the loop bounds are usually quite large. Therefore, the number of tiles in most dependence-free sets (after tiling) is also large. This provides us with the flexibility to select the most suitable set of tiles to schedule at each scheduling slot. In this paper, we take advantage of this flexibility, and schedule tiles in a BLP-aware fashion. Note that we are not proposing a tile shape/size selection strategy. Tile shape/size is usually dictated by cache locality and (loop-level) parallelism concerns, which are beyond our paper. Our proposed tile scheduling strategy works with any tile shape/size.

To our knowledge, except for [113], which focuses on single-core machines, all prior tile scheduling strategies considered the parallelism only from the program perspective, and did not take the bank-level parallelism into account. Conse-
sequently, it is not clear how tiling affects BLP and whether a BLP-aware tile scheduling can bring additional performance improvements over a (default) scheduling that does not care about BLP. Consider Figure 8.2(c) again: if $T_0$ through $T_3$ happen to access a very small number of banks, significant memory stalls can be experienced. In contrast, if these tiles access different banks, we may have high levels of BLP. Motivated by this, the problem we address in this paper can be defined as follows:

How can a compiler schedule the tiles in a dependence-free set such that high levels of BLP can be achieved?

Our proposed BLP-aware tile scheduling scheme can be used with any existing tiling strategy, and it consists of two steps. Figure 8.3 illustrates where our proposed compiler support fits in the relevant part of the compilation flow. In this first step, which is called Per-Tile Cache Miss Prediction, we predict the iterations in each tile that will incur last-level cache misses, by employing the Cache Miss Equations [114]. Then, in the second step, which is called Exploiting Bank-Level Parallelism, we determine the set of memory banks that will be accessed in each tile (based on the loop indices of these iterations, the base address, and the cache/memory parameters), and propose two algorithms that handle the cases of small and large tile sizes, respectively, to exploit BLP. With a small tile size, the sequence of accessed memory banks in each tile will not affect the scheduling decision much since these bank accesses can be considered “concurrent”. That is, as long as two tiles have the same memory bank to be accessed, scheduling these two tiles at the same logical time slot will hurt BLP since we have multiple accesses to the same memory bank around the same time. In contrast, with a large tile size, the duration (distance) between the first bank access and the last bank access in each tile could be long. As a result, the bank access order, in turn, affects the scheduling decision to maximize BLP. For example, even if two tiles have the same bank to be accessed, as long as the duration between the first access and second access to this bank is long enough (such that the first access is finished by the time when the second access starts), scheduling these two tiles at the same time slot can still achieve a high BLP.
Bibliography


[34] BOULET, P. ET AL. (1994) “(Pen)-ultimate tiling?” Integr. VLSI J.


[54] “Open64,” http://www.open64.net.


[64] Ding, C. and K. Kennedy (1999) “Improving cache performance in dynamic applications through data and computation reorganization at run time,” PLDI.


[72] Darte, A. et al. (2000) “Scheduling the computations of a loop nest with respect to a given mapping,” Euro-Par.


Vita
Wei Ding

Education

Ph.D. Candidate, Computer Science and Engineering, Pennsylvania State University - University Park, PA, USA

2009 – Present

M.S., Computer Integrated Manufacturing, Nanyang Technological University, Singapore

2007 – 2009

B.Eng., Software Engineering, Wuhan University, Wuhan, China

2003 – 2007

Selected Publications


• Wei Ding and Mahmut Kandemir. CApRI: CAche-conscious data Reordering for Irregular codes. In Proceedings of International Conference on ACM SIGMETRICS. (SIGMETRICS 2014)

