DATA LAYOUT OPTIMIZATION TECHNIQUES FOR MODERN AND EMERGING ARCHITECTURES

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of The Ohio State University

By

Qingda Lu, B.E., M.S.

The Ohio State University

2008

Dissertation Committee:
Dr. P. Sadayappan, Adviser
Dr. Xiaodong Zhang
Dr. J. Ramanujam
Dr. Atanas Rountev

Approved by

Adviser
Graduate Program in
Computer Science and
Engineering
ABSTRACT

The never-ending pursuit of higher performance is one fundamental driving force of computer science research. Although the semiconductor industry has fulfilled Moore’s Law over the last forty years by doubling transistor density every two years, the effectiveness of hardware advances cannot be fully exploited due to the mismatch between the architectural environment and the user program. Program optimization is a key to bridge this gap.

In this dissertation, instead of restructuring programs’ control flow as in many previous efforts, we have applied several new data layout optimization techniques to answer many optimization challenges on modern and emerging architectures. In particular, the developed techniques and their unique contributions are as follows.

- We describe an approach where a class of computations is modeled in terms of constituent operations that are empirically measured, thereby allowing modeling of the overall execution time. The performance model with empirically determined cost components is used to perform data layout optimization in the context of the Tensor Contraction Engine, a compiler for a high-level domain-specific language for expressing computational models in quantum chemistry.

- To obtain a highly optimized index permutation library for dynamic data layout optimization, we develop an integrated optimization framework that addresses a number of issues including tiling for the memory hierarchy, effective handling of memory
misalignment, utilizing memory subsystem characteristics, and the exploitation of the parallelism provided by the vector instruction sets in current processors. A judicious combination of analytical and empirical approaches is used to determine the most appropriate optimizations.

• With increasing numbers of cores, future CMPs (Chip Multi-Processors) are likely to have a tiled architecture with a portion of shared L2 cache on each tile and a bank-interleaved distribution of the address space. Although such an organization is effective for avoiding access hot-spots, it can cause a significant number of non-local L2 accesses for many commonly occurring regular data access patterns. We develop a compile-time framework for data locality optimization via data layout transformation. Using a polyhedral model, the program’s localizability is determined by analysis of its index set and array reference functions, followed by non-canonical data layout transformation to reduce non-local accesses for localizable computations.

• We leverage software and operating system utilities to identify locality patterns of data objects and allocate them accordingly with different priorities in caches. This data object locality guided caching strategy is mainly designed to address the inability of LRU replacement to effectively handle memory intensive programs with weak locality (such as streaming accesses) and contention among strong locality data objects in caches, so that sub-optimal replacement decisions can be avoided. To achieve our goal, we present a system software framework. We first collect object-relative reuse distance histograms and inter-object interference histograms via memory trace sampling. With several low-cost training runs, we are able to determine the locality
patterns of data objects. For the actual runs, we categorize data objects into different locality types and partition the cache space among data objects with a heuristic algorithm, in order to reduce cache misses through segregation of contending objects. The object-level cache partitioning framework has been implemented through modification of a Linux kernel.
Dedicated to my wife and my parents
ACKNOWLEDGMENTS

First and foremost, I am deeply indebted to my advisor Professor P. Sadayappan. Professor Sadayappan has been a great mentor in teaching me how to formulate problems, conduct research and present results. He showed me how to collaborate with others in research, and provided me with much freedom in finding interesting problems. His patience, compassion and hardworking spirit have been a great source of inspiration to me. I’m fortunate to have had the opportunity to work with and learn from him through the last six years.

I want to express my sincere gratitude to Professor Xiaodong Zhang for his guidance and help. Professor Zhang opened the door for me to study the interaction between computer architecture, operating systems and programming languages. He has always treated me as one of his own students. He spent many long hours with me on research discussion, introduced the best collaborators to me and shared with me his work and life experience. I have greatly benefited from him on both technical and personal matters.

I thank Professors J. Ramanujam, Atanas Rountev, Gerald Baumgartner and Zhao Zhang for having helped me greatly during my PhD study. I am very fortunate to have worked with them. Professors Ramanujam and Rountev have spent time serving on my
committee and provided valuable suggestions to improve this dissertation. Professor Baumgartner taught me how to build compilers and has always been a considerate friend. Professor Zhang educated me on many computer architecture issues and spent much time revising our collaborative papers.

I thank the current and previous members of Professor Sadayappan’s research group: Chi-chung Lam, Sandhya Krishnan, Albert Hartono, Xiaoyang Gao, Sriram Krishnamoorthy, Alina Bibireata, Daniel Cociorva, Alexandre Sibiriakov, Rajkiran Panuganti, Muthu Baskaran, Uday Bondhugula, Gaurav Khanna, Jim Dinan, Brian Larkins, Gerald Sabin and Thomas Henretty. Chi-chung and Daniel were the earliest members of the TCE team and started the project. Alina, Alex, Sandhya and Xiaoyang built TCE’s infrastructure on which a large part of my dissertation work is based. Sriram came up with numerous smart ideas and revised our collaborative papers sentence by sentence. Albert has been a good friend and a good listener. I enjoyed discussions with Raj, Muthu, Uday and Gaurav on research, cultures and news. Brian, Jerry, Jim and Tom provided many helpful insights on living in the U.S.. Special thanks to Xiaoyang for having offered me great help in many aspects of my life at OSU, from taking me to shop groceries in my first year in Columbus to giving suggestions on my job search.

During my PhD career, I had the privilege to work with many colleagues not from our research group, both inside and outside OSU. Jiang Lin and Xiaoning Ding have been great collaborators and friends. Though extremely busy, Jiang was always willing to spend time with me on technical and personal issues. Without Xiaoning’s help on Linux kernel debugging, this dissertation could not have been finished. Marcel Nooijen and his students helped me understand NWChem’s code; this helped me find many challenging problems such as optimizing index permutation.
Last, but not the least, I thank my family. I could never have finished this dissertation without their love and support. My deepest thanks to my wife Ye for her enduring love and constant encouragement. I thank my daughter Zihe for the joy and pride that she has brought to me. I thank my parents for their infinite love and patience. They have always encouraged me to follow my own plan and never asked about my graduation date. I also thank my parents-in-law for treating me as their own son. My mother-in-law helped us through my daughter’s first several months so I could focus on this dissertation.
VITA

1999 ................................................. B.E. Computer Science & Engineering
Beijing Institute of Technology, Beijing, China

2002 ................................................. M.S. Computer Science & Technology
Peking University, Beijing, China

2002-present ..................................... Graduate Research Associate,
The Ohio State University.

PUBLICATIONS

Research Publications

1. *Gaining Insights into Multi-Core Cache Partitioning: Bridging the Gap between Simulation and Real Systems*, Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, P. Sadayappan. In proceedings of the 14th International Symposium on High-Performance Computer Architecture (HPCA), 2008


4. *Performance modeling and optimization of parallel out-of-core tensor contractions*, Xiaoyang Gao, Swarup Kumar Sahoo, Qingda Lu, Chi-Chung Lam, J. Ramanujam, Gerald Baumgartner, P. Sadayappan. In proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2005


**FIELDS OF STUDY**

Major Field: Computer Science and Engineering
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>v</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>vi</td>
</tr>
<tr>
<td>Vita</td>
<td>ix</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xiv</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xv</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Program Optimization Challenges on Modern and Future Architectures</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 Memory Performance Problem</td>
<td>2</td>
</tr>
<tr>
<td>1.1.2 CMP Challenges</td>
<td>3</td>
</tr>
<tr>
<td>1.1.3 Additional Architectural Complexities</td>
<td>5</td>
</tr>
<tr>
<td>1.2 Solution through Data Layout Optimization</td>
<td>5</td>
</tr>
<tr>
<td>1.2.1 Motivating Examples</td>
<td>6</td>
</tr>
<tr>
<td>1.2.2 Advantages and Limitations</td>
<td>7</td>
</tr>
<tr>
<td>1.2.3 Design Choices</td>
<td>9</td>
</tr>
<tr>
<td>1.3 Dissertation Overview</td>
<td>11</td>
</tr>
<tr>
<td>2. Empirical Performance-Model Driven Data Layout Optimization for Tensor Contractions</td>
<td>13</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>13</td>
</tr>
<tr>
<td>2.2 The Computational Context</td>
<td>16</td>
</tr>
</tbody>
</table>
## Contents

### Chapter 2

- 2.3 Constituent Operations ........................................... 18
  - 2.3.1 Generalized Matrix Multiplication (GEMM) .................. 19
  - 2.3.2 Index Permutation .......................................... 22
- 2.4 Empirical measurement of constituent operations ............... 25
- 2.5 Composite Performance Model .................................. 27
- 2.6 Experimental Results ........................................... 30
- 2.7 Related Work ..................................................... 33
- 2.8 Summary .......................................................... 34

### Chapter 3

- 3. Combining Analytical and Empirical Approaches in Tuning Index Permutation 36
  - 3.1 Introduction .................................................... 36
  - 3.2 Background and Motivation ................................... 38
  - 3.3 Input Parameters ............................................... 41
  - 3.4 Cache Hierarchy Optimizations ................................ 44
    - 3.4.1 Tiling for Spatial Locality ............................... 44
    - 3.4.2 Two-level Tiling to Handle Cache Misalignment .......... 45
    - 3.4.3 Minimizing TLB Misses .................................. 46
    - 3.4.4 Identifying and Handling Conflict Misses ................ 47
  - 3.5 Memory Subsystem Optimizations ............................... 49
    - 3.5.1 Buffering of Writes .................................... 49
    - 3.5.2 Loop Order Selection .................................... 50
  - 3.6 Exploiting SIMD instruction sets ................................ 50
  - 3.7 Integrated Optimization Framework ............................ 52
  - 3.8 Experimental Results ......................................... 55
  - 3.9 Extending to High Dimensions .................................. 62
  - 3.10 Related Work ................................................... 64
  - 3.11 Summary ....................................................... 67

### Chapter 4

- 4. Data Layout Transformation for Enhancing Data Locality on NUCA Chip Multiprocessors 68
  - 4.1 Introduction .................................................. 68
  - 4.2 Localization Analysis .......................................... 74
    - 4.2.1 Polyhedral Framework for Program Transformation ........ 74
    - 4.2.2 Localization Constraints .................................. 76
    - 4.2.3 Localization Analysis Algorithm .......................... 76
  - 4.3 Data Layout Transformation .................................... 80
  - 4.4 Code Generation ................................................ 84
  - 4.5 Evaluation ..................................................... 88
  - 4.6 Related Work ................................................... 93
  - 4.7 Summary ....................................................... 96
5. Improving Hardware Cache Performance Through Software-Controlled Object-Level Cache Partitioning ............................... 97  
   5.1 Introduction .............................................. 97  
   5.2 A Motivating Example ................................. 100  
   5.3 Overview of the Approach ............................. 102  
   5.4 Object-Level Program Locality Profile ................. 104  
      5.4.1 Modeling Object-Relative Temporal Locality ...... 104  
      5.4.2 Modeling Inter-Object Interference ............... 107  
      5.4.3 Cache Miss Estimation .............................. 109  
   5.5 Profile Generation ...................................... 111  
   5.6 Profile Analysis ......................................... 114  
   5.7 Cache Partitioning Decision Making and Enforcement 115  
      5.7.1 Partitioning Enforcement ......................... 117  
      5.7.2 Partitioning Decision Making ..................... 118  
   5.8 Experimental Results ................................... 121  
   5.9 Related Work ........................................... 131  
   5.10 Summary ............................................... 134  

6. Conclusion and Future Work .................................. 136  
   6.1 Contributions of the Dissertation ..................... 136  
   6.2 Future Directions ....................................... 138  

Bibliography ................................................. 141
<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Configuration of the Itanium 2 cluster at OSC</td>
</tr>
<tr>
<td>2.2</td>
<td>Layouts and distributions for the CCSD computation for the unoptimized and optimized versions of the code</td>
</tr>
<tr>
<td>2.3</td>
<td>Sequential performance results for ccsd and 4index-transform</td>
</tr>
<tr>
<td>2.4</td>
<td>Parallel performance results on 4 processors for ccsd and 4index-transform</td>
</tr>
<tr>
<td>3.1</td>
<td>Optimization challenges for matrix transposition</td>
</tr>
<tr>
<td>3.2</td>
<td>Architectural parameters and configuration used for evaluation</td>
</tr>
<tr>
<td>3.3</td>
<td>Analytically-determined parameters for single-precision matrix transposition</td>
</tr>
<tr>
<td>3.4</td>
<td>Empirically determined parameters for single-precision matrix transposition</td>
</tr>
<tr>
<td>4.1</td>
<td>Simulation Configuration</td>
</tr>
<tr>
<td>4.2</td>
<td>A data-parallel benchmark suite used in evaluation.</td>
</tr>
<tr>
<td>4.3</td>
<td>Localization analysis results of the benchmarks.</td>
</tr>
<tr>
<td>5.1</td>
<td>Categorizing objects in CG’s profiles.</td>
</tr>
<tr>
<td>5.2</td>
<td>Characteristics and experimental results for selected benchmarks. For brevity 0,1,2,3 are used to represent constant, linear, square and cubic functions respectively.</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A motivating example for employing data layout optimization</td>
<td>7</td>
</tr>
<tr>
<td>1.2</td>
<td>Another motivating example for employing data layout optimization</td>
<td>8</td>
</tr>
<tr>
<td>2.1</td>
<td>Operator tree for a sub-expression in the CCSD equation. (a) Original</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>operator tree (b) The expanded operator tree used for optimal code generation</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>The processing steps in the extended Cannon Algorithm. Initially processor</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>$P_{ij}$ holds blocks labeled $B_{ij}$ and $A_{i(j,j+1)}$. The portion of</td>
<td></td>
</tr>
<tr>
<td></td>
<td>data accessed in each step is shown in bold</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>The matrix multiplication times using the MKL library for $C(M, N) :=$</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>$A(M, K) \ast B(K, N)$ where $K = N = 4000$. $M$ is varied along the x-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>axis. The performance obtained in shown on the y-axis in GFLOPS. (a)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>transb='t' (b) transb='n' in input argument to dgemm</td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>The matrix multiplication times using the ATLAS library for $C(M, N) :=$</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>$A(M, K) \ast B(K, N)$ where $K = N = 4000$. $M$ is varied along the x-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>axis. The performance obtained in shown on the y-axis in GFLOPS. (a)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>transb='t' (b) transb='n' in input argument to dgemm</td>
<td></td>
</tr>
<tr>
<td>2.5</td>
<td>Index permutation times for three different permutations for an $N \times$</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>$N \times N \times N \times N$ matrix using (a) Intel Fortran compiler</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(b) g77 compiler. $N$ is varied along the x-axis. The y-axis shows the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>execution time per double word in clock cycles</td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>A simple implementation of matrix transposition</td>
<td>39</td>
</tr>
<tr>
<td>3.2</td>
<td>Improvement from using SIMD in memory copy</td>
<td>42</td>
</tr>
</tbody>
</table>
3.3 Two-level tiling to handle cache misalignment with tiling factor $T=2$. The arrow indicates strided inter-tile loop order ........................................ 47

3.4 Library code generation procedure ........................................... 53

3.5 Matrix transposition code with one-level tiling and buffering writes .... 54

3.6 Runtime decision tree for implementation selection. The leaves of the tree correspond to the implementations ........................................... 55

3.7 Effect of adding optimization considerations on (a) Pentium 4 and (b) PowerPC G5 ................................................................. 58

3.8 Performance of different versions with cache-aligned arrays on (a) Pentium 4 and (b) PowerPC G5 ................................................... 59

3.9 Performance of different versions with cache-misaligned arrays on (a) Pentium 4 and (b) PowerPC G5 ................................................... 61

3.10 Normalized execution time of triples correction with different molecule inputs on Pentium 4 ......................................................... 65

3.11 Normalized execution time of CCSDT with different molecule inputs on Pentium 4 ................................................................. 66

4.1 (a) Tiled CMP architecture. (b) Mapping a physical address to a cache bank on a tiled CMP ......................................................... 70

4.2 1D Jacobi code. .............................................................. 71

4.3 Parallelizing a 1D Jacobi program on a 4-tile CMP ......................... 71

4.4 Matrix-vector multiplication code ........................................... 79

4.5 A one-dimensional example of data transformations. (a) Canonical layout after strip-mining. (b) Non-canonical layout after strip-mining and permutation ......................................................... 83

4.6 Normalized remote L2 cache access numbers with $\#_{remoteAccesses}(canon) = 1$ ................................................................. 92
4.7 Normalized link utilization of on-chip network with \( \text{utilization}(\text{canon}) = 1 \) 92

4.8 Speedup relative to code version \textit{canon}. ............................... 93

5.1 An outline of NAS-CG code. .................................................. 101

5.2 Overall structure of the object-level cache partitioning framework. ...... 103

5.3 Object-relative reuse distance histograms for objects \( p \) and \( a \) in CG, generated with input class B. .................................................. 105

5.4 Examples of inter-object interference histograms for CG, generated with input class B. .......................................................... 108

5.5 The combined reuse distance histogram for object group \( \{a, p\} \) in CG with input class B. .......................................................... 110

5.6 Program profile generation with a training input. ............................ 111

5.7 Cache partitioning decision making and enforcement with an actual input. 116

5.8 CG with object-level cache partitioning in comparison to uncontrolled LRU caching. .......................................................... 128

5.9 LU with object-level cache partitioning in comparison to uncontrolled LRU caching. .......................................................... 129

5.10 \textit{art} with partial and complete object-level cache partitioning in comparison to standard LRU caching. .......................................................... 131
CHAPTER 1

INTRODUCTION

1.1 Program Optimization Challenges on Modern and Future Architectures

The never-ending pursuit of higher performance is a fundamental driving force of computer science research. Although the semiconductor industry has fulfilled Moore’s Law over the last forty years, doubling transistor density and processor speed every two years, the demand for higher computing speed has only increased. In scientific computing, it is not uncommon to run a program for days or weeks on a massively parallel computer with thousands of processors; in computer architecture research, it takes tens of hours to simulate several seconds of a program execution; in commercial computing, server farms consisting of tens of thousands of servers are deployed by service providers such as Google to offer their users instant responses.

To address this urgency for performance, computer architects and designers have made numerous hardware advances. However, the effectiveness of these techniques usually cannot be fully exploited due to the mismatch between the architectural environment and user programs. Program optimization is therefore needed to transform programs or data to find a better match.
1.1.1 Memory Performance Problem

The performance gap between the processor and DRAM has been increasing exponentially for over two decades. On a modern processor, an off-chip memory access often costs hundreds of clock cycles. This “memory wall” problem is likely to persist due to the limited off-chip bandwidth [17]. By providing fast data and instruction buffers to on-chip computation resources, caching has been the most widely used technique to bridge the performance gap between the processor and DRAM. A cache is organized as a set of cache blocks (lines) whose typical sizes range from 16 bytes to 128 bytes on modern architectures. If a data item is buffered in cache, adjacent data in the same cache line are also loaded. A memory reference has a *cache hit* if the requested data is in a cache block; otherwise it incurs a *cache miss*, and the data is loaded from memory.

Exploitation of the principle of *locality of reference* is essential for high performance. There are three basic types of reference locality and caches attempt to translate them to cache hits.

- *Temporal locality*: A memory location that is referenced at one point in time is likely to be again referenced soon in the future. Cache hits from temporal locality are said to exploit *temporal cache reuse*.

- *Spatial locality*: The likelihood of referencing a memory location is higher if another memory location near it has been referenced. Cache hits from spatial locality utilize *spatial cache reuse*.

- *Sequential locality*: Memory is accessed sequentially or following a non-unit constant stride. In the literature, sequential locality has often been viewed as a special
form of spatial locality. In this dissertation, we treat them differently in that sequential locality reveals the patterns of long memory access streams which enable linear prefetching of instructions and data. If a cache hit is due to sequential locality and not reuse, it is referred to as a *prefetch hit*.

There has recently been an increasing trend towards the analysis of large data sets, conducting dynamic computation and using modularized programming [38]. In many modern software systems, accesses to the same data item are far separated in time, and data accesses are often non-contiguous, with large non-constant strides. With these problems, temporal locality cannot be effectively exploited since a data item is often evicted from cache before being reused; spatial locality is often weak, which wastes the cache’s effective capacity; and automatic strided prefetching is not possible due to the lack of sequential locality. Therefore this software trend has made efficient utilization of cache resources an increasingly challenging problem.

### 1.1.2 CMP Challenges

Power and heat constraints have made it extremely difficult to further improve performance through complex and power-hungry single-threaded processor designs. Instead computer architects have integrated multiple simple processor cores on a single chip to increase throughput and reduce power consumption. This trend introduces new challenges and makes software program optimization much more important in achieving high performance on chip-multiprocessors (CMPs).

**Resource Contention**

Physically distributed processor cores on a CMP have their private computation resources such as ALU. However certain on-chip resources are shared among processor cores
to maximize their utilization. The competition for these shared resources can lead to destructive inter-thread interference.

The most commonly shared resources are the last levels of on-chip cache and the memory subsystem. In this proposal, we focus on shared L2 caches that have been available in many existing commercial processors [113, 56, 66] and used in proposals for future CMPs [51, 11]. Multiprogrammed workloads contend for both cache capacity and associativity. Problems such as cache thrashing from inter-thread interference have been reported by previous studies [43, 65]. Program optimization has to answer this challenge with a software-only approach or through interaction with additional hardware support.

**Growing Wire Delay**

The technological trend towards diminutive feature sizes has made on-chip wire delay a critical bottleneck in achieving high performance. Proposed approaches employ a Non-Uniform Cache Architecture (NUCA) design that spreads data across cache banks and connects these banks through an on-chip network [64, 51, 11]. Although NUCA was originally proposed for uniprocessors [64], its target has followed the technical trend towards CMPs. Because on average only a small portion of cached data may be close to a processor core, careful data placement is critical to reducing memory access latencies.

**Requirement for Parallelization**

Historically, parallelism has been extracted dynamically by processors at the instruction level. The advent of chip multi-processing demands parallelism at the thread level to utilize the large number of on-chip resources without paying excessive synchronization and communication costs. The process of parallelization typically involves identifying independent tasks within the program and assigning these tasks to threads. Manual parallelization is
error-prone and requires the understanding of the whole program. Therefore it is highly desirable to develop automatic program parallelization frameworks that maximize parallelism while controlling resource contention and data access costs.

### 1.1.3 Additional Architectural Complexities

In addition to the above challenges, many modern architectural features further complicate program optimization. Two of these complexities are as follows.

**Exploitation of SIMD Extensions**

Many modern processors have adopted multimedia extensions characterized by Single Instruction Multiple Data (SIMD) units operating on packed short vectors. Examples of these SIMD extensions include SSE/SSE2/SSE3 for Intel processors and VMX/Altivec for PowerPC processors. SIMD extensions often provide an independent register file and require strict data alignments in memory. When possible, SIMD instructions should be used to replace scalar instructions to achieve higher performance.

**Exploitation of Memory-level Parallelism**

Techniques exploiting instruction-level parallelism attempt to hide memory stall times by using miss information/status handling registers (MSHRs) to keep track of cache misses. As demonstrated in [90], due to the limited size of the instruction window on an out-of-order processor, memory accesses have to be carefully scheduled to effectively overlap cache misses.

### 1.2 Solution through Data Layout Optimization

To answer the challenges of program optimization on modern and future architectures, our proposed optimization techniques focus on transformations in two spaces: computation
and data. Computation optimization is more commonly used and transforms the computation structure of a program to match its execution environment. Examples of computation transformations include loop transformations for locality enhancement [120, 86] and automatic loop parallelization [97, 75].

An alternative approach is to transform programs’ data layouts. In this dissertation, we apply data layout optimization to improve program performance in different contexts that represent architectural and software trends. Combined with computation transformations, our data layout optimization techniques are able to answer many architectural challenges that cannot be fully addressed by computation transformations alone.

1.2.1 Motivating Examples

Here we illustrate the basic idea and the effectiveness of data layout optimization with the following motivating examples.

1. Consider the C program in Figure 1.1. While arrays A and C are accessed sequentially, array B is read with a large stride. The conflicting access patterns among these three arrays make it difficult to increase data locality with linear loop transformations such as permutation. While loop tiling is useful in this case to reduce cache misses, selection of tile sizes is still a challenging task. Furthermore, when N is a large power of two, even loop tiling is not useful due to the large number of conflict misses from both inter-array and intra-array accesses.

By allowing data layout transformations, one easy solution is to transpose the layout of array B. In this way all three arrays are accessed sequentially with the optimal data locality and the only code transformation needed is to change B’s indexing function from $[j][i]$ to $[i][j]$. 
for (i = 0; i < N; i++)
for (j = 0; j < N; j++)
    C[i][j] = A[i][j] + B[j][i];

Figure 1.1: A motivating example for employing data layout optimization

2. Another example to consider is shown in Figure 1.2. In the first loop, objects in two different vectors are created in an interleaved fashion. After this creation process the vectors are scanned separately for objects satisfying different conditions. Assuming the size of a ClassA object is significantly less than the cache line size and each vector is large enough, in the two scanning loops half of the memory traffic is wasteful since it fetches unused data into cache. Due to data dependences between two allocation sites, loop fission cannot be applied to the creation loop. Similarly the dynamic properties of this program such as runtime loop bounds may prevent fusing the two scanning loops to increase spatial locality.

An efficient solution to this problem is through data layout optimization. By grouping objects of each vector on the heap based on their allocation sites, data brought into cache during the scanning loops can be fully used. To support this optimization, the two allocation sites need to be distinguished with annotations to the memory management system.

1.2.2 Advantages and Limitations

Advantages of Data Layout Optimization

Data layout and computation transformations complement each other. The most important advantage of data layout optimization is that it does not change memory access orders
for (i = 0; i < N; i++) {
    ClassA* pObj1 = new ClassA(i);
    ClassA* pObj2 = new ClassB(pObj1);
    vec1.insert(pObj1);
    vec2.insert(pObj2);
}

for (i = 0; i < vec1.size(); i++) {
    ClassA* tempObj1 = vec1.get(i);
    if (test1(tempObj1)) {
        tempObj1->update();
        break;
    }
}

for (i = 0; i < vec2.size(); i++) {
    ClassA* tempObj2 = vec2.get(i);
    if (test2(tempObj2)) {
        tempObj2->update();
        break;
    }
}

Figure 1.2: Another motivating example for employing data layout optimization

therefore no data dependence can be violated. Instead the applicability of many computation transformations is constrained by data dependences. As shown by the two examples in Section 1.2.1, in many cases where computation transformations are hard to apply, there exist simple and efficient optimization solutions through data layout transformations.

Limitations of Data Layout Optimization

There are two important limitations that restrict the applicability of data layout optimization.
1. Data layout optimization has a global effect across the whole program. Although certain data layouts may be optimal during one program phase, they may be suboptimal for other program phases. This limitation requires understanding of the whole program through either accurate static inter-procedural analysis or dynamic analysis of whole-program execution traces. In comparison, the effect of computation transformations is mostly local.

2. The applicability of data layout optimization may rely on the type safety of the programming language. Lower-level languages such as C and C++ have many type-unsafe features such as unsafe type casting and pointer arithmetic operations. These features hamper many data layout transformations such as field reordering and structure splitting [31].

1.2.3 Design Choices

There is a large design space to explore in order to apply efficient data layout optimization within a given hardware and software environment.

**Data Layout Transformation-Only vs. An Integrated Approach**

One fundamental design choice is to consider data layout optimization only or have an integrated approach that employs both computation and data layout transformations. The latter approach is more general and we attempt to employ it when possible. However, for certain problems such as that in Chapter 2, only focusing on data layouts can abstract away complex low-level details and we choose to only optimize data layouts in these cases.
Intra-Object Layout vs. Inter-Object Layout

It is useful to consider the layout of a data object at two levels of granularity. At a fine granularity, we need to decide how to organize data within an object. For an array object, its intra-object layout refers to the approach to map an index to a memory location. For a structure with data members, its intra-object layout refers to the relative placement of each data member. At a coarser granularity, inter-object layout refers to the placement of objects in the memory space.

As shown in Section 1.2.1, data layout optimization considers both intra- and inter-array layout. In this dissertation, we focus on intra-array layout in Chapters 2, 3 and 4 and optimize inter-object layout in Chapter 5.

Static Layout Selection vs. Dynamic Layout Transformation

Data layouts can be decided and used throughout a program’s execution or they can be changed dynamically to overcome the global effect of data layout optimization. There are different ways of adjusting object data layouts at run time. The compiler can insert data reorganization code between program phases or controlled runtime environments such as a Java virtual machine can move heap objects. The complexity of transforming data layouts dynamically lies in the difficulty of assuring program correctness and avoiding the runtime transformation overhead. In this dissertation, we consider dynamic approaches and also offer efficient runtime support for array restructuring.

Analytical Model vs. Empirical Search

Architectural complexities make it difficult to find the best one from many transformed versions of a program without running them with actual inputs. Despite the additional
search cost, empirical search has been used in optimizing many performance-critical pro-
grams [29, 117, 81]. In this dissertation we attempt to combine analytical models and  
empirical search to apply data layout optimization and tune data transformation libraries.

Canonical Array Layout vs. Non-Canonical Array Layout

An important observation is that there is often a mismatch between an array’s standard 
row-major or column-major layout and the underlying architecture. Instead array elements  
can be laid out in the memory following non-canonical layouts such as Z-Morton [28].  
When index functions are properly optimized, indexing costs associated with non-canonical  
arrays are often amortized by the overall performance improvement. In this dissertation,  
non-canonical layout array layouts are employed in Chapter 4 to overcome large wire de-
lays on NUCA processors.

1.3 Dissertation Overview

In this dissertation, we have developed a new set of data layout optimization techniques  
to help unleash the computing power of modern and emerging architectures for several  
important problems.

This dissertation is organized as follows. Chapter 2 uses data layout optimization in  
the context of the Tensor Contraction Engine, a compiler for a high-level domain-specific  
language for expressing computational models in quantum chemistry. Our optimization is  
based on a performance model with empirically determined costs of constituent operations.  
Chapter 3 presents the optimization of index permutation routines, an important group of  
layout transformation kernels. To tolerate the effect of growing wire delays, Chapter 4  
proposes a framework that employs non-canonical data layouts to improve data locality  
on NUCA chip multiprocessors. Chapter 5 presents a software framework that partitions
the cache at the object level, in order to improve program performance for both single-threaded and parallel data-sharing programs. We conclude in Chapter 6 by summarizing the contributions of this dissertation and pointing out future research directions.
In this chapter, we present how data layout optimization is applied in a domain-specific compiler – Tensor Contraction Engine (TCE) in order to improve the performance of both sequential and parallel programs.

2.1 Introduction

Optimizing compilers use high-level program transformations to generate efficient code. The computation is modeled in some form and its cost is derived in terms of metrics such as reuse distance. Program transformations are then applied to the computational model to minimize its cost. The large number of parameters and the variety of programs to be handled limits optimizing compilers to model-driven optimization with relatively simple cost models. Approaches to empirically optimize a computation, such as ATLAS [117], generate solutions for different structures of the optimized code and determine the parameters that optimize the execution time by running different versions of the code and choosing the optimal one. But empirical optimization of large complex applications can be prohibitively expensive. In this chapter, we decompose a class of computations into its
constituent operations and model the execution time of the computation in terms of empirical characterization of its constituent operations. The empirical measurements allow modeling of the overall execution time of the computation while decomposition enables offline determination of the cost model and efficient global optimization across multiple constituent operations.

Our domain of interest is the calculation of electronic structure properties using ab initio quantum chemistry models such as the coupled cluster models [71]. We have developed an automatic synthesis system called the Tensor Contraction Engine (TCE), to generate efficient parallel programs from high-level expressions, for a class of computations expressible as tensor contractions [10, 36, 35]. These calculations employ multi-dimensional tensors in contractions, which are essentially generalized matrix multiplications. The computation is represented by an operator tree, in which each node represents the contraction of two tensors to produce a result tensor. The order of indices of the intermediate tensors (multidimensional arrays) is not constrained.

Computational kernels such as Basic Linear Algebra Subroutines (BLAS) [39] have been tuned to achieve very high performance. These hand-tuned or empirically optimized kernels generally achieve better performance than conventional general-purpose compilers [125]. If General Matrix Multiplication (GEMM) routines available in BLAS libraries are used to perform tensor contractions, the multi-dimensional intermediate arrays that arise in tensor contractions must be transformed to group the indices to allow a two-dimensional view of the arrays, as required by GEMM. We observe that the performance of the GEMM routines is significantly influenced by the choice of parameters used in their invocation. We determine the layouts of the intermediate arrays and the choice of parameters to the GEMM invocations that minimize the overall execution time. The overall execution time
is estimated from the GEMM and index permutation times. Empirically-derived costs for these constituent operations are used to determine the GEMM parameters and array layouts.

The approach presented in this chapter may be viewed as an instance of the telescoping languages approach described in [63]. The telescoping languages approach aims at facilitating a high-level scripting interface for a domain-specific computation to the user, while achieving high performance that is portable across machine architectures, and compilation time that only grows linearly with the size of the user script. In this chapter, we evaluate the performance of the relevant libraries empirically. Parallel code is generated using the Global Arrays (GA) library [87]. Parallel matrix multiplication is performed using the Cannon matrix multiplication algorithm [20], extended to handle non-square distribution of matrices amongst the processors. The matrix multiplication within each node is performed using GEMM. The parallel matrix multiplication and parallel index transformation costs are estimated from the local GEMM and transformation costs and the communication cost. We then use the empirical results to construct a performance model that enables the code generator to determine the appropriate choice of array layouts and distributions and usage modalities for library calls.

This chapter is organized as follows. In Section 2.2, we elaborate on the computational context, demonstrate potential optimization opportunities and then define our problem. Section 2.3 discusses the constituent operations in the computation and the parameters to be determined to generate optimal parallel code. Section 2.4 describes the determination of the constituent operation costs. Section 2.5 discusses the determination of the parameters of the generated code from the constituent operation costs. Results are presented in Section 2.6. Section 2.7 discusses related work. Section 2.8 summarizes the chapter.
2.2 The Computational Context

The Tensor Contraction Engine (TCE) [10, 36, 35] is a domain-specific compiler for developing accurate \textit{ab initio} models in quantum chemistry. The TCE takes as input a high-level specification of a computation expressed as a set of tensor contraction expressions and transforms it into efficient parallel code. In the class of computations considered, the final result to be computed can be expressed as multi-dimensional summations of the product of several input arrays.

Consider the following tensor contraction expression.

\[ E[i, j, k] = \text{Sum}\{a, b, c\} A[a, b, c] B[a, i] C[b, j] D[c, k] \]

where all indices range over \( N \). \( a, b, c \) are the summation indices. The direct way to compute this would require \( O(N^6) \) arithmetic operations. Instead, by computing the following intermediate partial results, the number of operations can be reduced to \( O(N^4) \).

\[
T_1[a, b, k] = \text{Sum}\{c\} A[a, b, c] D[c, k]
\]

\[
T_2[a, j, k] = \text{Sum}\{b\} T_1[a, b, k] C[b, j]
\]

\[
E[i, j, k] = \text{Sum}\{a\} T_2[a, j, k] B[a, i]
\]

This form of the computation is represented as an operator tree. For example, Figure 2.1(a) shows the operator tree for a sub-expression from the CCSD (Coupled Cluster Singles and Doubles) model [71]. The curly braces around the indices indicate the fact that there is no implied ordering between the indices. The computation represented by such an operator tree could be implemented as a collection of nested loops, one per node of the operator tree. However, optimizing the resulting collection of a large number of nested loops to minimize
execution time is a difficult challenge. But each contraction is essentially a generalized matrix multiplication, for which efficient tuned library Generalized Matrix Multiplication (GEMM) routines exist. Hence it is attractive to translate the computation for each tensor contraction node into a call to GEMM. For the above 3-contraction example, the first contraction can be implemented directly as a call to GEMM with $A$ viewed as an $N^2 \times N$ rectangular matrix and $D$ as an $N \times N$ matrix. The second contraction, however, cannot be directly implemented as a GEMM call because the summation index $b$ is the middle index of $T1$. GEMM requires the summation indices and non-summation indices in the contraction to be collected into two separate contiguous groups. In order to use GEMM, $T1$ needs to be “reshaped”, e.g. $T1[a, b, k] \rightarrow T1_{r}[a, k, b]$. Then GEMM can be invoked with the first operand $T1_{r}$ viewed as an $N^2 \times N$ array and the second input operand $C$ as an $N \times N$ array. The result, which has the index order $[a, k, j]$, would have to be reshaped to give $T2[a, j, k]$. Since $T2$ is only a temporary intermediate array, the order of its dimensions could be chosen to be $[a, k, j]$ instead of $[a, j, k]$, which avoids the need to reshape the output array produced by GEMM. Considering the last contraction, it might seem that some reshaping would be necessary in order to use GEMM. However, GEMM allows one or both of its input operands to be transposed. Thus, the contraction can be achieved by invoking GEMM with $B$ as the first operand in transposed form, and $T2[a, j, k]$ as the second operand, with shape $N \times N^2$. But, as will be shown later, the performance of GEMM for transposed and non-transposed input operands could differ significantly.

In general, a sequence of multi-dimensional tensor contractions can be implemented using a sequence of GEMM calls, possibly with some additional array reordering operations interspersed. We represent this computation as an expanded operator tree. For example, Figure 2.1(b) shows the expanded operator tree derived from the operator tree in
Figure 2.1: Operator tree for a sub-expression in the CCSD equation. (a) Original operator tree (b) The expanded operator tree used for optimal code generation

Figure 2.1(a). Each node in the operator tree is replicated to represent a possible array reordering. The problem addressed in this chapter is: given a sequence of tensor contractions (expressed as an expanded operator tree), determine the layout (i.e., dimension order) and distribution (among multiple processors) of the tensors, and the modes of invocation of GEMM so that the specified computation is executed in minimal time.

2.3 Constituent Operations

In this section we discuss various operations within the computation and their influence on the execution time. The parameters that influence these costs, and hence the overall execution time, are detailed.
2.3.1 Generalized Matrix Multiplication (GEMM)

General Matrix Multiplication (GEMM) is a set of matrix multiplication subroutines in the BLAS library. It is used to compute

\[ C := \alpha \cdot \text{op}(A) \cdot \text{op}(B) + \beta \cdot C. \]

In this chapter, we use the double precision version of the GEMM routine of the form

\[ \text{dgemm}(\text{transa}, \text{transb}, m, n, k, \alpha, A, lda, B, ldb, \beta, C, ldc), \]

where \( \text{transa} \) (\( \text{transb} \)) specifies whether \( A \) (\( B \)) is in the transposed form. When \( \text{transa} \) is 'n' or 'N', \( \text{op}(A) = A \); when \( \text{transa} \) equals to 't' or 'T', \( \text{op}(A) = A^T \); \( \alpha \) and \( \beta \) are scalars; \( C \) is an \( M \times N \) matrix; \( \text{op}(A) \) and \( \text{op}(B) \) are matrices of dimensions \( M \times K \) and \( K \times N \), respectively.

We measured the variation in the performance of GEMM with variation in its input parameters on the Itanium 2 Cluster at the Ohio Supercomputer Center (Dual 900 MHz processors with 4 GB memory, interconnected by Myrinet 2000 network). The cluster’s configuration is shown in Table 2.1. The latency and bandwidth measurements of the interconnect were obtained from [4]. Matrix multiplications of the form \( A \ast B \) were performed, where \( B \) was a \( 4000 \times 4000 \) matrix and \( A \) was an \( M \times 4000 \) matrix, with \( M \) varied from 1 to 300. Matrix multiplications involving such oblong matrices is quite typical in quantum chemistry computations. Two BLAS libraries available in the Itanium 2 Cluster, ATLAS

<table>
<thead>
<tr>
<th>Node</th>
<th>Memory</th>
<th>OS</th>
<th>Compilers</th>
<th>TLB</th>
<th>Network Latency</th>
<th>Interconnect</th>
<th>Comm. Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dual 900MHz</td>
<td>4GB</td>
<td>Linux 2.4.21smp</td>
<td>g77, ifc</td>
<td>128 entry</td>
<td>17.8 µs</td>
<td>Myrinet 2000</td>
<td>ARMCI</td>
</tr>
</tbody>
</table>

Table 2.1: Configuration of the Itanium 2 cluster at OSC
and the Intel Math Kernel Library (MKL) [53] were evaluated. The \textit{transb} argument was specified as \textquote{t} for the results shown in Figure 2.3(a) and Figure 2.4(a). Figure 2.3(b) and Figure 2.4(b) show the results for \textit{transb} being \textquote{n}. The x-axis shows the value of \(M\) and the y-axis represents the performance of matrix multiplication in GFLOPS. We observe that the performance of the GEMM operation for the transposed and untransposed versions cannot be interpreted as the cost of transposition at the beginning of the computation for the experiments with transposed \(B\). For example, in some of the experiments with the ATLAS library, the transposed version performs better. Thus the parameters of the DGEMM invocations need to be determined so as to optimize the overall execution time.

\section*{Cannon’s Matrix Multiplication Algorithm}

Several approaches have been proposed for implementing parallel matrix multiplication [46, 39]. In this chapter, we consider an extension to Cannon’s algorithm [20], which removes the restriction of using a square grid of processors for array distribution.

The extended Cannon algorithm for a \(4 \times 2\) processors grid is illustrated for the matrix multiplication \(C(M, N) := A(M, K) \ast B(K, N)\) in Figure 2.2. The processors form a logical rectangular grid. All the arrays are distributed amongst the processors in the grid in an identical fashion. Each processor holds a block of arrays \(A\), \(B\) and \(C\). The algorithm divides the common dimension (\(K\) in this illustration) to have the same number of sub-blocks. Each step operates on a sub-block and not on the entire data local to each processor. In each step, if the sub-block required is local to the processor, no communication is required. Figure 2.2 shows in bold the sub-blocks of arrays \(A\) and \(B\) accessed in each step. It shows that the entire \(B\) array is accessed in each step.
Figure 2.2: The processing steps in the extended Cannon Algorithm. Initially processor $P_{ij}$ holds blocks labeled $B_{ij}$ and $A_{i(j:j+1)}$. The portion of data accessed in each step is shown in bold.

Given a processor grid, the number of steps is given by the number of sub-blocks along the common dimension ($K$). The number of blocks of $A$ that are needed by one processor corresponds to the number of processors along the common dimension, and that of $B$ correspond to the other dimension. The number of steps and the number of remote blocks required per processor depend on the distribution of the arrays amongst the processors. The block size for communication is independent of the dimensions. It can be seen that different distributions have different costs for each of the components.
Figure 2.3: The matrix multiplication times using the MKL library for $C(M, N) \leftarrow A(M, K) \ast B(K, N)$ where $K = N = 4000$. $M$ is varied along the x-axis. The performance obtained is shown on the y-axis in GFLOPS. (a) transb='t' (b) transb='n' in input argument to dgemm.

The relative sizes of the arrays $A$ and $B$ determine the optimal distribution. When one array is much larger than the other, the cost can be reduced by skewing the distribution to reduce the number of remote blocks accessed for that array. The shape of the array that is local to each processor affects the local DGEMM cost. Thus, the array distribution influences the communication and computation costs and is an important parameter to be determined.

### 2.3.2 Index Permutation

DGEMM requires a two-dimensional view of the input matrices. This means that the summation and non-summation indices of a tensor contraction must be grouped into two contiguous sets of indices. Thus the layout of a multi-dimensional array might have to be transformed to be used as input to DGEMM. Further, additional index permutation cost
might be worth paying if it can reduce the DGEMM cost through the use of a transposed (or non-transposed) argument form.

We implemented a collection of index permutation routines, one each for a different number of dimensions. The routines were tiled in the fastest varying indices in the source and target arrays. We observed that performing the computation such that the target arrays are traversed in the order of their storage resulted in better performance than biasing the access to the source array. The execution times for different tile sizes was determined and the best tile size was chosen. The performance of the routines was evaluated on a number of permutations to determine the tile sizes.

We measured the execution times of these routines for some index permutations on four-dimensional arrays of size $N \times N \times N \times N$, with $N$ varying from 15 to 85. The results are shown in Figure 2.5. Different permutations are observed to incur different costs. We also notice that the use of different compilers leads to different performances.
Figure 2.5: Index permutation times for three different permutations for an $N \times N \times N \times N$ matrix using (a) Intel Fortran compiler (b) g77 compiler. $N$ is varied along the x-axis. The y-axis shows the execution time per double word in clock cycles.

The layout of the arrays influences the index permutation costs and is the parameter to be determined to evaluate the index permutation cost. Parallel index permutation can be viewed as a combination of local index permutation and array redistribution. The extended Cannon’s algorithm requires that the summation and non-summation index groups be distributed along the slowest varying index in that group. The number of processors along the dimension in the processor grid corresponding to a group can also be varied to determine the shape/size of arrays used in the local DGEMM calls. Thus, in addition to the layout of the arrays, their distribution needs to be determined as well.

Note that the layout of input and output arrays for a DGEMM invocation uniquely determines its parameters. Thus the problem of determination of the DGEMM parameters can be reduced to the layout optimization problem. The variation in the cost of DGEMM with its parameters has the effect of increasing the search space to be explored.
2.4 Empirical measurement of constituent operations

**GEMM cost**

The DGEMM cost can be determined by executing the DGEMM routine with the specified parameters on the target machine. Alternatively, the cost of DGEMM routines in a library on a particular system can be modeled by sampling it offline. The DGEMM cost for the relevant parameters can then be estimated from the sampled data set. Executing the DGEMM at runtime increases the code generation time, while estimating it leads to potential inaccuracies in the cost model.

In this chapter, we determine the cost of DGEMM by executing it with the specified parameters. In the operator tree model considered, in which each non-leaf node requires exactly one invocation of the DGEMM, this could result in compilation times that are as long as the execution times. But in real quantum chemistry computations, which require out-of-core treatment [67], tiles of multi-dimensional arrays are brought into memory and operated upon. These loops are in turn enclosed in an outermost loop in iterative chemical methods. Thus each node in the operator tree requires multiple invocations of DGEMM. Therefore the compilation time is much less than the execution time.

**Cannon’s Matrix Multiplication**

The cost of parallel matrix multiplication using Cannon’s algorithm is the sum of the computation and the communication costs. Since the local computation is performed using DGEMM, the computation cost can be derived from the DGEMM cost. The communication cost is the sum of the communication costs at the beginning of each step. A latency-bandwidth model is used to determine the communication cost. Consider the matrix multiplication $C(M, N) += A(M, K) \ast B(K, N)$. Let $P_M, P_K, P_N$ be the number of processors...
into which the array is distributed along the $M$, $N$ and $K$ dimensions, respectively. The total communication cost is given by

$$CommnCost = CommnCost_A + CommnCost_B$$

$$CommnCost_A = (T_s + \frac{M \times K}{BW \times P_M \times P_K}) \times (P_K - P_K/P_M)$$

$$CommnCost_B = (T_s + \frac{K \times N}{BW \times P_K \times P_N}) \times (P_K - P_K/P_N)$$

where $T_s$ is the latency of the interconnect shown in Table 2.1. $BW$, the bandwidth is estimated from a table constructed from the bandwidth curve in [4].

**Index Permutation**

Figure 2.5 shows the performance of our index permutation routines for some permutations. The performance of the implementation appears to be relatively independent of the array dimensions, but is influenced by the permutation being performed.

An analysis of the implementation revealed that the variation in the per-element permutation cost was primarily influenced by the variation in the TLB misses for different permutations and the capability of compilers to perform efficient register tiling.

We estimated the index permutation cost to consist of two components. The first component is the basic copy cost, the minimum cost required to copy a multi-dimensional array, together with the index calculation. We determined two types of basic copy costs. The first, referred to as $c_0$, is the one in which both the source and target arrays are traversed to have sufficient locality. The other basic copy cost, referred to as $c_1$, is one in which only the target array is traversed to have locality. Depending on the permutation and the size of the arrays, one of these basic copy costs is chosen. Note that with multi-level tiling of the routines there would be only one basic copy cost. The basic costs $c_0$ and $c_1$ were found to be compiler dependent. They were determined to be 9.5 and 11.3 cycles, respectively.
per double word with the Intel Fortran Compiler and 12.9 and 15.9 cycles, respectively, per double word with g77. The second component is the TLB miss cost. Each processor on the Itanium-2 cluster had an 128 entry fully-associative TLB with a miss penalty of 25 cycles. Different permutations can lead to different blocks of data being contiguously accessed and at different strides. The permutation to be performed and the array size are used to determine the TLB cost.

In the parallel version of the algorithm, index permutation is coupled with array redistribution. Transformation from one layout and distribution configuration to another is accomplished in two steps, a local index permutation followed by array redistribution.

A combination of index permutation and redistribution can result in each processor communicating its data to more than one processor. The communication cost is estimated differently for different cases. When the target patch written to is local to a processor no communication is required. When the layout transformation is such that each processor needs to communicate its data to exactly one other processor, the cost is uniform across all the processors and is estimated as the cost of communicating that block. In all other cases, we estimate the communication cost to be the cost incurred by the processor whose data is scattered among the most number of processors.

### 2.5 Composite Performance Model

In this section, we discuss how the empirical measurements of the constituent operations are used to determine the parameters that optimize the overall execution time.
The input and output arrays are constrained to have one layout each. The feasible layouts for the other nodes is given by the following equation.

\[
S(n) = \begin{cases} 
\bigcup (\forall l \in \mathcal{P}(NSI(n'))(\forall r \in \mathcal{P}(NSI(n'')))\{(l, r) \cup (r, l)\} & \text{if } n \text{ is a contraction node} \\
\bigcup (\forall l \in \mathcal{P}(SI(n'))(\forall r \in \mathcal{P}(NSI(n')))\{(l, r) \cup (r, l)\} & \text{if } n \text{ is an index permutation node}
\end{cases}
\]

where \(S(n)\) is the set of possible layouts, \(SI(n)\) the set of summation indices and \(NSI(n)\) the set of non-summation indices in node \(n\). \(\mathcal{P}\) is the set of all permutations of its argument. \(n'\) and \(n''\) are the left and right child of node \(n\), respectively. \((l, r)\) denotes the concatenation of the sequences \(l\) and \(r\).

A tree node \(C\) corresponding to a DGEMM computation of the form \(C(M, N) += A(M, K) \ast B(K, N)\) can have layouts corresponding to the cross-product of the permutations of the non-summation indices of its children. The remaining nodes are index permutation nodes and are constrained by the layouts acceptable by their parent (i.e., the contraction node to which they act as input). They have layouts corresponding to the cross-product of the permutations of their summation and non-summation indices.

For example, if \(A\) and \(B\) contain 3 non-summation and 2 summation indices (as determined by the \(C\) array) each, \(A\) and \(B\) have \(3! \ast 2! \ast 2 = 24\) possible layouts each and \(C\) has \(3! \ast 3! \ast 2 = 72\) possible layouts.

The extended Cannon algorithm requires all distributions to be rectangular in nature. In addition, the children of each contraction node in the operator tree are required to have the same distribution as that node. Thus, for each distribution of a contraction node, there is a corresponding distribution for its children. There is no restriction on the distribution of the contraction nodes themselves.
Determination of Optimal Parameters

For the specified layout and distribution of the root and leaves of the operator tree, we determine the layouts and distributions of the intermediate arrays. For each layout of an array produced by DGEMM, the arrays corresponding to its children nodes are required to have a compatible layout, i.e. the order in which the summation and non-summation indices are grouped is required to be identical in the produced and consumed arrays. This is because the DGEMM does not perform any index permutation within a group. This restriction is used to prune candidate layouts.

The configuration of an array is represented by a layout-distribution pair. Dynamic programming is used to determine the optimal configuration for the intermediate arrays. The cost of a node is determined as the least cost to compute its children and subsequently compute it from its children. It is as follows.

\[
C_t(n, d, l) = \begin{cases} 
\min_{d', l' \in \mathcal{D}, l'' \in \mathcal{L}} C_t(n', d', l') + C_{ip}((n', d', l') \rightarrow (n, d, l)) & \text{if } n \text{ is a index permute node} \\
\min_{l' \in \mathcal{L}} C_t(n', d, l') + C_t(n'', d, l'') + C_{dg}((n', d, l') \times (n'', d, l'') \rightarrow (n, d, l)) & \text{if } n \text{ is a contraction node}
\end{cases}
\]

where

\[
\begin{align*}
C_t & \equiv \text{Total cost of computing a node with relevant } (d, l) \\
C_{ip} & \equiv \text{Cost of the required index permutation} \\
C_{dg} & \equiv \text{Cost of the required DGEMM invocation} \\
\mathcal{D}(\mathcal{L}) & \equiv \text{All feasible distributions (layouts) of relevant node} \\
n'(n') & \equiv \text{Left(Right) child of } n
\end{align*}
\]

The expanded operator tree for the example in Figure 2.1(a) is shown in Figure 2.1(b). The replicated nodes correspond to the index permutations. The original nodes correspond...
to the contractions. Thus, in the expanded operator tree, each non-leaf node is computed from its children by either index permutation or contraction. Therefore, the total cost of computation of each non-leaf, for different configurations, can be determined from the cost of computing its children from the leaf nodes and the cost of the basic operation, index permutation or GEMM, to compute the node from its children. The algorithm first determines the feasible layouts for each of the nodes in the expanded operator tree. The optimal cost of the root node is subsequently computed using the dynamic programming formulation described above.

### 2.6 Experimental Results

We evaluated our approach on the OSC Itanium-2 cluster whose configuration is shown in Table 2.1. All the experiment programs were compiled with the Intel Itanium Fortran Compiler for Linux. We considered two computations in our domain.

1. CCSD: We used a typical sub-expression from the CCSD theory used to determine electronic structures.

   \[
   S(j, i, b, a) = \text{Sum}\{l, k\} (A\{l, k, b, a\} \times (\text{Sum}\{d\} (\text{Sum}\{c\} (B\{d, c, l, k\} \times C\{i, c\}) \times D\{j, d\})))
   \]

   All the array dimensions were 64 for the sequential experiments and 96 for the parallel experiments.

2. AO-to-MO transform: This expression, henceforth referred to as the 4-index transform, is commonly used to transform two-electron integrals from atomic orbital (AO) basis to molecular orbital (MO) basis.

   \[
   B(a, b, c, d) = \text{Sum}\{s\} (C1\{s, d\} \times \text{Sum}\{r\} (C2\{r, c\} \times \text{Sum}\{q\} (C3\{q, b\} \times \text{Sum}\{p\} (C4\{p, a\} \times A\{p, q, r, s\}))))
   \]
<table>
<thead>
<tr>
<th>Array</th>
<th>Distribution</th>
<th>Dist. Index</th>
<th>Layout</th>
<th>GEMM. Parameters</th>
<th>Distribution</th>
<th>Dist. Index</th>
<th>Layout</th>
<th>GEMM. Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>(2,2)</td>
<td>(k,a)</td>
<td>(l,k,b,a)</td>
<td>–</td>
<td>(1,4)</td>
<td>(k,a)</td>
<td>(l,k,b,a)</td>
<td>–</td>
</tr>
<tr>
<td>A’</td>
<td>(2,2)</td>
<td>(a,k)</td>
<td>(b,a,l)</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>(2,2)</td>
<td>(k,c)</td>
<td>(d,l,c)</td>
<td>–</td>
<td>(1,4)</td>
<td>(c,k)</td>
<td>(d,c,l)</td>
<td>–</td>
</tr>
<tr>
<td>B’</td>
<td>(2,2)</td>
<td>(i,c)</td>
<td>(c,i)</td>
<td>–</td>
<td>(1,4)</td>
<td>(i,c)</td>
<td>(i,c)</td>
<td>–</td>
</tr>
<tr>
<td>C</td>
<td>(2,2)</td>
<td>(j,d)</td>
<td>(j,d)</td>
<td>–</td>
<td>(1,4)</td>
<td>(j,d)</td>
<td>(j,d)</td>
<td>–</td>
</tr>
<tr>
<td>D</td>
<td>(2,2)</td>
<td>(i,d)</td>
<td>(d,j)</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>(2,2)</td>
<td>(k,i)</td>
<td>(d,l,i)</td>
<td>B’.C’,(‘n’,‘n’)</td>
<td>(1,4)</td>
<td>(k,i)</td>
<td>(d,l)</td>
<td>C,B’,(‘n’,‘n’)</td>
</tr>
<tr>
<td>T1’</td>
<td>(2,2)</td>
<td>(i,j)</td>
<td>(l,k,i)</td>
<td>T1’,D’,(‘n’,‘n’)</td>
<td>(1,4)</td>
<td>(j,k)</td>
<td>(j,l)</td>
<td>D,T1’,(‘n’,‘n’)</td>
</tr>
<tr>
<td>T2</td>
<td>(2,2)</td>
<td>(k,i)</td>
<td>(l,k,i)</td>
<td>–</td>
<td>(1,4)</td>
<td>(i,k)</td>
<td>(i,l)</td>
<td>–</td>
</tr>
<tr>
<td>T2’</td>
<td>(2,2)</td>
<td>(a,j)</td>
<td>(b,a,j)</td>
<td>A’,T2,(‘n’,‘n’)</td>
<td>(4,1)</td>
<td>(a,i)</td>
<td>(b,a,j)</td>
<td>A,T2, (‘t’,‘t’)</td>
</tr>
<tr>
<td>S</td>
<td>(2,2)</td>
<td>(a,j)</td>
<td>(j,i,b,a)</td>
<td>–</td>
<td>(1,4)</td>
<td>(i,a)</td>
<td>(j,i,b)</td>
<td>–</td>
</tr>
<tr>
<td>S’</td>
<td>(2,2)</td>
<td>(i,a)</td>
<td>(j,i,b,a)</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: Layouts and distributions for the CCSD computation for the unoptimized and optimized versions of the code
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GEMM</td>
<td>Index</td>
<td>Exec.</td>
<td>GFLOPS</td>
<td></td>
<td>GEMM</td>
<td>Index</td>
</tr>
<tr>
<td></td>
<td>Perm.</td>
<td>Perm.</td>
<td>Time</td>
<td></td>
<td></td>
<td>Perm.</td>
<td>Perm.</td>
</tr>
<tr>
<td>ccbsd</td>
<td>55.28</td>
<td>1.41</td>
<td>56.69</td>
<td>2.50</td>
<td>45.58</td>
<td>0.78</td>
<td>46.36</td>
</tr>
<tr>
<td>4index</td>
<td>10.06</td>
<td>2.58</td>
<td>12.64</td>
<td>2.07</td>
<td>10.58</td>
<td>0.0</td>
<td>10.58</td>
</tr>
</tbody>
</table>

Table 2.3: Sequential performance results for ccbsd and 4index-transform

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GEMM</td>
<td>Index</td>
<td>Exec.</td>
<td>GFLOPS</td>
<td></td>
<td>GEMM</td>
<td>Index</td>
</tr>
<tr>
<td></td>
<td>Perm.</td>
<td>Perm.</td>
<td>Time</td>
<td></td>
<td></td>
<td>Perm.</td>
<td>Perm.</td>
</tr>
<tr>
<td>ccbsd</td>
<td>157.93</td>
<td>7.21</td>
<td>165.14</td>
<td>9.68</td>
<td>136.41</td>
<td>2.86</td>
<td>139.27</td>
</tr>
<tr>
<td>4index</td>
<td>12.23</td>
<td>7.74</td>
<td>19.97</td>
<td>3.27</td>
<td>7.57</td>
<td>3.64</td>
<td>11.21</td>
</tr>
</tbody>
</table>

Table 2.4: Parallel performance results on 4 processors for ccbsd and 4index-transform

The array dimensions were 80 and 96 for the sequential and parallel experiments.

We compared our approach with the baseline implementation in which an initial layout for the arrays is provided. A fixed $\sqrt{P} \times \sqrt{P}$ array distribution is required throughout the computation. This approach was, in fact, used in our early implementations. The optimized version is allowed flexibility in the distribution of the input and output arrays.

Table 2.2 shows the configurations chosen for each array in the parallel experiment for the unoptimized and optimized cases. A first look reveals that the number of intermediate arrays is reduced by effective choice of layouts and distributions. The GEMM parameters for all three GEMM invocations is different, either in the order chosen for the input arrays or in the transposition of the input parameters. The distribution chosen for all the arrays is different from those for the unoptimized version of the computation.
Table 2.3 and Table 2.4 show the sequential and parallel results respectively. In the parallel experiments, the GEMM and index permutation times reported subsume the communication costs. The optimized version has close to 20% improvement over the unoptimized version in almost all cases. The parallel 4-index transform has an improvement of more than 75% over the unoptimized version. The effective choice of GEMM parameters results in a noticeable improvement in the GEMM cost for most cases. The index permutation cost is either improved or totally eliminated. The trade-off between the GEMM and the index permutation costs can be observed in the sequential 4-index transform experiment. In this experiment, the optimization process chooses an inferior configuration for the GEMM computation, so as to eliminate the index permutation cost completely, and hence reduce the overall execution time.

2.7 Related Work

There has been prior work that has attempted to use data layout optimizations to improve spatial locality in programs, either in addition to or instead of loop transformations. Leung and Zahorjan [72] were the first to demonstrate cases where loop transformations fail (for a variety of reasons) for which data transformations are useful. The data transformations they consider correspond to non-singular linear transformations of the data space. O’Boyle and Knijnenburg [89] present techniques for generating efficient code for several layout optimizations such as linear transformations memory layouts, alignment of arrays to page boundaries, and page replication. Several authors [8, 55] discuss the use of data transformations to improve locality on shared memory machines. Kandemir et al. [60] present a hyperplane representation of memory layouts of multi-dimensional arrays and show how
to use this representation to derive very general data transformations for a single perfectly-nested loop. In the absence of dynamic data layouts, the layout of an array has an impact on the spatial locality characteristic of all the loop nests in the program which access the array. As a result, Kandemir et al. [58, 59, 60] and Leung and Zakhirjan [72] present a global approach to this problem; of these, [58] considers dynamic layouts.

Some authors have addressed unifying loop and data transformations into a single framework. These works [33, 59] use loop permutations and array dimension permutations in an exhaustive search to determine the appropriate loop and data transformations for a single nest and then extend it to handle multiple nests.

FFTW [81] and ATLAS [117] produce high performance libraries for specific computation kernels, by executing different versions of the computation and choosing the parameters that optimize the overall execution time. Our approach is similar to these in that we perform empirical evaluation of the constituent operations for various possible parameters. But our work focuses on a more general class of computations than a single kernel. This forbids an exhaustive search strategy.

2.8 Summary

We have described an approach to the synthesis of efficient parallel code that minimizes the overall execution time. The approach was developed for a program synthesis system targeted at the quantum chemistry domain. The code was generated as a sequence of DGEMM calls interspersed with index permutation and redistribution to enable to use of the BLAS libraries and to improve overall performance. The costs of the constituent operations in the computation were empirically measured and were used to model the cost of the computation. This computational model was used to determine layouts and distributions
that minimize the overall execution time. Experimental results were provided that showed the effectiveness of our approach.
CHAPTER 3

COMBINING ANALYTICAL AND EMPIRICAL APPROACHES IN TUNING INDEX PERMUTATION

Index permutation routines are arguably the most important layout transformation kernels. In this chapter, we present how we tune index permutation combining both analytical and empirical approaches on different architectures. We first focus on two-dimensional index permutation – matrix transposition and then we extend our approach to higher dimensional cases.

3.1 Introduction

Index permutation kernels are a group of important kernels occurring in many applications. Although they essentially involve only memory copying, the required permutation results in complex problem-size dependent access patterns. It is a challenging task to develop index permutation code that is optimized for different array sizes and alignments on different architectures.

In this chapter, we first address the development of a library for high-performance matrix transposition then we extend our approach to multi-dimensional index permutations. We identify the optimization opportunities and parameters in matrix transposition. We then
employ both offline analysis and empirical search to determine the appropriate optimizations. When the exact optimization choices cannot be made at installation time, we generate multiple versions and choose the appropriate version at runtime.

We address various issues and optimization opportunities in achieving high-performance for matrix transposition. We present a two-level tiling strategy to improve spatial locality in the L1 cache. An analytical model to minimize TLB misses is presented. A model is developed to predict the presence of conflict misses for a subset of the problem instances. For other cases, we develop a low-cost inspector that can accurately detect the presence of conflict misses at runtime. Mechanisms to take into consideration other characteristics of the memory subsystem are included. We take advantage of the SIMD instruction sets widely available in modern computing systems. They have been shown to be beneficial in improving the memory bandwidth of bandwidth-limited computations such as memory copy. The parallelism in the SIMD hardware also enables efficient register-level matrix transposition. In exploiting these features, we address attendant issues such as data alignment.

A special code generator has been designed and implemented to automatically generate code using SSE/SSE2 on Intel architectures, Altivec on PowerPC architectures, or just scalar instructions.

We show that the the cost of matrix transposition can be reduced significantly using our approach. The effectiveness of our approach in tuning for the target architecture is demonstrated by the performance improvement obtained in matrix transposition using just the scalar instruction set. Further optimizations using the SIMD instruction set result in the best performance of matrix transposition to be up to 43% and 98% of memcpy provided by the native operating system, on Pentium 4 and PowerPC G5, respectively.
The chapter is organized as follows: Section 3.2 provides background on the matrix transposition problem. Section 3.3 elaborates on the matrix transposition problem and defines its input parameters. Optimizations targeting the memory hierarchy are addressed in Section 3.4. Section 3.5 details optimizations targeting the characteristics of modern memory subsystems. Exploitation of SIMD instruction sets is described in Section 3.6. Section 3.7 presents the integrated framework that combines the various optimizations described to develop a matrix transposition library for a given target system. Experimental results of matrix transposition are presented in Section 3.8. Section 3.9 presents how the optimization procedure is extended to high-dimensional index permutations and experimental results. Section 3.10 discusses related work and Section 3.11 concludes the chapter.

3.2 Background and Motivation

Matrix transposition is widely used in many types of applications and is one of the most fundamental array operations. Despite its ubiquitous nature, matrix transposition is still commonly written in loop form as shown in Figure 3.1. Existing compilers are incapable of translating this implementation into an efficient executable code. For example, the program in Figure 3.1 was compiled using the Intel C compiler with “-O3” option. On an Intel Pentium 4 with a 533MHz front side bus, the executable achieved an average bandwidth of 90.3MB/s, for single-precision arrays with each dimension ranging from 3800 to 4200. This is only 4.4% of the sustained copy bandwidth reported by the STREAM memory benchmark [85].

Similar to many streaming multimedia workloads, matrix transposition lacks temporal locality (each element is accessed once) and has a large cache footprint. The data access
1) for $i = 0$ to $N1-1$
2) for $j = 0$ to $N2-1$
3) $B[i][j] = A[j][i]$

Figure 3.1: A simple implementation of matrix transposition

pattern in matrix transposition makes it difficult to avoid performance problems due to the cache hierarchy. In particular, the following issues need to be addressed:

- In the absence of temporal locality, only spatial locality can be exploited to reduce cache misses.

- The strided access pattern potentially incurs high conflict miss penalties, due to the limited associativity of caches.

- For large array sizes, the strided access leads to heavy TLB miss penalties.

Matrix transposition can be characterized as a memory bandwidth-bound problem. A substantial portion of the total overhead is attributable to the memory subsystem (the memory bus and the main memory) in the following aspects:

- Since the output array never needs to read, a cache hierarchy with write-allocate policy wastes memory bandwidth by bringing the destination array into cache.

- Dead cycles between back-to-back read/write transactions are needed with many memory-bus designs. Previous studies [15] have demonstrated that significant bandwidth loss can be caused by memory-bus turnaround. Since matrix transposition consists of an equal number of read and write operations, there can be a significant loss of memory bandwidth even after careful optimization for the cache hierarchy.
• At the DRAM side, a row of data on a memory bank is read into a row buffer, providing an opportunity to exploit locality. A memory schedule may reuse the data in a row buffer efficiently eliminating additional precharge and row activation. In addition, accesses to data on different memory banks can be pipelined in popular DRAM designs.

Many modern processors have adopted multimedia extensions characterized as Single Instruction Multiple Data (SIMD) units operating on packed short vectors. Examples of these SIMD extensions include SSE/SSE2/SSE3 for Intel processors and VMX/Altivec for PowerPC processors. When SIMD support is employed, programs such as matrix transposition can benefit from the following aspects:

• SIMD extensions often provide an independent large register file to reduce cache pressure. Data reorganization instructions provide an opportunity to efficiently reorganize data at register level.

• A SIMD load/store instruction often outperforms an equivalent sequence of scalar memory access instructions.

• Cacheability control is often provided by modern architectures as part of their multimedia extensions. The cacheability control instructions such as software prefetch and streaming writes enable software assisted data cache management and have a significant impact on streaming applications.

The above discussion is summarized in Table 3.1.

1Some cacheability control instructions are not part of multimedia extensions, e.g. instructions dcbt and dcbz in PowerPC.
<table>
<thead>
<tr>
<th>Category</th>
<th>Challenge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache hierarchy</td>
<td>Exploiting spatial locality</td>
</tr>
<tr>
<td></td>
<td>Minimizing conflict misses</td>
</tr>
<tr>
<td></td>
<td>Reducing TLB misses</td>
</tr>
<tr>
<td>Memory subsystem</td>
<td>Improving memory bandwidth utilization</td>
</tr>
<tr>
<td></td>
<td>Reducing memory-bus turnaround overhead</td>
</tr>
<tr>
<td></td>
<td>Exploiting efficient DRAM accesses</td>
</tr>
<tr>
<td>SIMD instructions</td>
<td>Exploiting intra-register data reorganization</td>
</tr>
<tr>
<td></td>
<td>Accessing data with alignment constraints</td>
</tr>
<tr>
<td></td>
<td>Exploiting cacheability control instructions</td>
</tr>
</tbody>
</table>

Table 3.1: Optimization challenges for matrix transposition

To illustrate the potential benefits of employing SIMD extensions in memory bandwidth-bound problems, Figure 3.2 shows the performance difference in memory copy performance using the scalar and SIMD instruction sets on an Intel Pentium 4 and a PowerPC G5. The configurations of these two processors are listed in Table 3.2.

### 3.3 Input Parameters

Our objective is to generate an efficient implementation of the matrix transposition operation. We use a combination of offline analysis and empirical search to determine the appropriate choice of optimization parameters. The empirical search happens at installation time, and is similar to the ATLAS approach to generating an efficient BLAS library [118, 117].

The following inputs, which define the architecture, are available to the code generator at compile-time:

- Cache sizes $C_1$ and $C_2$, in bytes, of L1 and L2 data caches, respectively.
- Cache line sizes $L_1$ and $L_2$, in bytes, for L1 and L2 caches, respectively.
• Degrees of cache associativity $K_1$ and $K_2$ of L1 and L2 caches, respectively.

• Vector size $S_v$ in bytes. We have $S_v = 16$ in both SSE/SSE2 and Altivec.

• Number of vector registers $N_r$.

• Number of TLB entries $N_{TLB}$ \(^2\)

In systems with an L3 cache, L3 cache parameters are used in place of that of L2. In general, we are interested in the innermost and outermost levels of the cache hierarchy. Two platforms are used in our research, an Intel Pentium 4 with SSE/SSE2 and an Apple G5 with Altivec. The configuration and architectural parameters of these two machines are listed in Table 3.2.

\(^2\)We do not address the limited associativity of TLB entries on some systems.
<table>
<thead>
<tr>
<th></th>
<th>Pentium 4 2.4GHz</th>
<th>PowerPC G5 2.0GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_1 )</td>
<td>8K</td>
<td>32K</td>
</tr>
<tr>
<td>( L_1 )</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>( K_1 )</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>( C_2 )</td>
<td>512K</td>
<td>512K</td>
</tr>
<tr>
<td>( L_2 )</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>( K_2 )</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>( S_v )</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>( N_r )</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>( N_{TLB} )</td>
<td>64</td>
<td>1024</td>
</tr>
<tr>
<td>Front-side Bus</td>
<td>533 MHz</td>
<td>1 GHz</td>
</tr>
<tr>
<td>OS</td>
<td>Linux 2.4</td>
<td>Mac OS X 10.4</td>
</tr>
<tr>
<td>Compiler</td>
<td>icc 8.0</td>
<td>Apple gcc 3.3</td>
</tr>
<tr>
<td>Compiler Flags</td>
<td>-O3</td>
<td>-O3 -faltivec</td>
</tr>
</tbody>
</table>

Table 3.2: Architectural parameters and configuration used for evaluation

The following inputs provide information on the specific matrix transposition required:

- Data element size: \( E \) bytes. Since scientific programs largely use either double-precision or single-precision floating point arrays, we consider \( E = 4 \) and \( E = 8 \) in this chapter. In this chapter, matrix transposition results are presented with \( E = 4 \), corresponding to single-precision arrays. Note that larger element sizes simplifies permutation and exploitation of locality.

- Array shape: \((N_1, N_2)\). In this work we consider row-major layout with the second dimension being the fastest varying dimension.

- Array base address: the base address of the source array \( Addr_{src} \) and the base address of the destination array \( Addr_{dst} \). Most architectures and compilers provide natural alignments, which means for any data element at address \( A \) with data size \( E \) bytes, it is guaranteed that \( A \mod E = 0 \).
An array is said to be cache aligned if its base address and number of elements in a row are a multiple of cache line size.

The code generator takes as input the architectural parameters and generates multiple versions of code optimized for different categories of problem instances.

For brevity, we introduce the following notation:

- Vector size in terms of number of elements $V_e = S_v/E$.
- $L_1$ Cache line size in terms of number of elements $L_{1e} = L_1/E$. Similarly, we have $L_{2e} = L_2/E$.

### 3.4 Cache Hierarchy Optimizations

In this section, we discuss the optimizations addressing the cache hierarchy in modern systems.

#### 3.4.1 Tiling for Spatial Locality

We use tiling to exploit spatial locality. The tile size is chosen to be a multiple of $L_{1e}$, as other tile sizes lead to accessing partial cache lines, potentially increasing the cache misses over a tile size that is smaller but an integral multiple of $L_{1e}$. A tile size of $L_{2e}$ is most beneficial as it helps bring each element into both L1 and L2 caches just once. Due to the absence of temporal locality, increasing the tile size beyond the $L_{2e}$ does not improve performance, while increasing the possibility of both conflict and capacity misses. We refer to the matrix transposition within this tile level as mini-transposition.

The intra-tile loop indices will henceforth be referred to as $i$ and $j$ while the inter-tile loop indices will be referred to as $iT$ and $jT$. Note that the row-major loop order ($iT,jT$) corresponds to contiguous access of the source array.
The appropriate tile sizes are chosen empirically from the range of \((L_{1e}, L_{1e})\) and \((L_{2e}, L_{2e})\), such that the tile fits in the L1 cache. Note that non-square tile sizes can also improve spatial locality, depending on the loop order. For example, given the loop order \((iT,jT)\) for the tiling loops, the tile size \((L_{1e}, L_{2e})\) allows the remainder of L2 cache lines of the source array to be accessed in the next tile, while improving spatial locality for the target array. The tile size along the fastest varying dimension for the destination array is denoted by \(\text{tilesize}_1\), and the tile size along the other dimension is denoted by \(\text{tilesize}_2\).

Note that tile size determination is problem-independent and can be performed once. When the architecture parameters do not dictate a particular tile size, we perform an empirical search of the tile sizes together with the associated loop order. The determination of the loop order is discussed in more detail in a subsequent section.

### 3.4.2 Two-level Tiling to Handle Cache Misalignment

The tiling described above can be insufficient when the arrays are not cache aligned. For example, consider the tiling shown in Figure 3.3. The tile size is chosen to be \(L_{2e}\), and the tiles are accessed in a strided fashion, as depicted by the downward arrow. For larger problem sizes, this would result in all L2 cache lines except one to miss in the L1 cache, as the entire column has to be scanned before the partially accessed cache lines can be reused. This effectively doubles the number of L1 cache misses over the minimum possible.

Simply increasing the tile size greatly increases chances of conflict and capacity misses. A reordering of the processing of the tiles, as indicated by the numbering of the tiles, allows retaining of the partially accessed cache lines thus enabling reuse. In the best case, this reduces the number of cache misses per tile to eight from fifteen. This reordering is achieved by another level of tiling.
The tile size for the second level is chosen to be a multiple of the L2 cache line size, $T \times L_{2e}$. If a cache line is accessed in the outer tile but not accessed in the inner tile, it should be kept in L1 cache until it is fully accessed. Therefore, we have the following observations based on Figure 3.3:

1. If the writes for an inner tile are not buffered, explained in the next section, the L1 cache needs to hold at least $(2T + 2)L_{2e}$ cache lines. Therefore, $1 \leq T \leq \frac{C_1}{2L_2L_{2e}} - 1$. We conservatively have $T = \frac{C_1}{2L_2L_{2e}} - 2$, to leave some L1 cache space for temporary variables.

2. Otherwise, only $(T + 2)L_{2e}$ tiles need to be kept in the L1 cache since the data buffer is guaranteed to be cache aligned. We have $1 \leq T \leq \frac{C_1}{L_2L_{2e}} - 2$. Similar to the above case, we choose $T = \frac{C_1}{L_2L_{2e}} - 3$.

### 3.4.3 Minimizing TLB Misses

TLB misses can be severe given the memory access pattern in matrix transposition, despite tiling for the cache hierarchy. Another level of tiling can be introduced to alleviate the TLB misses. We select this tile size to be

$$\text{tilesize}_{\text{TLB}} = \frac{N_{\text{TLB}}}{2} - \text{Max}(\text{tilesize}_1, \text{tilesize}_2).$$

This enables us to efficiently use about half the TLB entries without introducing further complexity of having conflict misses with in-cache buffers or worrying about the loop order for this outermost loop level.

On Pentium 4, which has $N_{\text{TLB}} = 64$, $\text{tilesize}_{\text{TLB}}$ is selected as 16, which is the same as $\text{tilesize}_1$ and $\text{tilesize}_2$, making tiling for TLB redundant. On PowerPC G5, because
3.4.4 Identifying and Handling Conflict Misses

Most existing literature on matrix transposition assumes the presence of conflict misses [21, 44]. While severe in the context of bit-reversal, matrix transposition of arrays with relatively large dimensions is not afflicted by conflict misses to the same degree. We simulated the conflict misses when the base addresses of the arrays are cache aligned and $N1$ and $N2$
are varied from 1 to $4 \times C_2 / E$. We found that conflict misses occur in the L1 cache for only 12.1% of the cases for the architecture parameters corresponding to Intel Pentium 4.

If both arrays are cache-aligned and both dimensions are multiples of $L_{2e}$, conflict misses can be predicted based on the following observations:

Let $N$ be represented as $2^n \times (2s + 1)$. Conflict misses can only occur in the following cases:

1. When $2^n < \frac{C_1}{K_1E}$ and $\frac{C_1}{2^nE} < \text{tilesize}_1$.

2. When $2^n \geq \frac{C_1}{K_1E}$.

When a tile in mini-transposition is not cache-aligned, any row in this tile of size $m \times L_{1e}$ crosses $m + 1$ L1 cache lines. In practice, we have $m = 1$ or $m = 2$. For an arbitrary base address and array shape, it is not easy to analytically determine the presence of conflict misses. However, since the tile size is a multiple of $L_{1e}$, it is guaranteed that the cache footprints of all tiles have the same pattern. Figure 3.3 illustrates such an example, where the cache line offset patterns in different tiles are identical and the distance between two neighboring tile rows is guaranteed to be the same across tiles. Since one row in a tile crosses at most 3 cache lines in a tile, we can inspect no more than $3 \times \text{tilesize}_1$ cache lines to determine whether there are conflict misses in one array. This inspection procedure has a very low overhead since it is only invoked once for all of the tiles in an array. Note that most modern processors have a virtually indexed L1 cache. Our inspection procedure is accurate on Pentium 4 because any given cache line equivalent of data in an array can only be mapped to a single L1 cache set (4 cache lines). In general, our procedure is inaccurate on processors with physically tagged L1 cache, such as PowerPC G5, because contiguity in cache lines depends on the virtual-to-physical mapping. Zhang and Zhang [126] observed
that contiguous locations in virtual memory are usually mapped to contiguous locations in
physical memory. Under this “contiguous allocation” assumption, our model is accurate.

We handle conflict misses by using buffers to copy the data corresponding to the in-
nermost tile before processing. Depending on the order of the loops, one of the arrays is
accessed contiguously, one row at a time. The other array is buffered to handle conflict
misses. The effective detection of conflict misses is critical, since a conservative assump-
tion of the presence of conflict misses would unnecessarily introduce copying, impairing
performance. On the other hand, not detecting conflict misses can significantly worsen
performance.

3.5 Memory Subsystem Optimizations

3.5.1 Buffering of Writes

Most modern architectures provide instruction-level mechanisms that enable cacheabil-
ity control. For example, AltiVec provides instructions to zero out a cache line, without
any read or write of the corresponding address in memory. This can be utilized to eliminate
unnecessary bandwidth in reading a write-only array into cache. Going further, SSE/SSE2
provide non-temporal write instructions that write an entire cache line, stored in a sequence
of SIMD registers, directly into memory. This not only avoids wastage of bandwidth read-
ing write-only data into cache, but also reduces cache pollution reducing the possibility of
conflict and capacity misses.

Both mechanisms require the grouping of operations on the write array to operate on
full cache lines. When tiling is used, in order to write a complete cache line, a buffer of size
least $L_{1e}$ elements is needed. For single-precision arrays, such an in-register buffer would
require 16 registers on the Intel Pentium 4 and 32 registers on the PowerPC G5. Since there
are only 8 SIMD registers on Pentium 4 and 32 SIMD registers on PowerPC G5, which are also needed for read and permutation operations, it is not feasible to maintain a buffer using the available registers. To overcome this limitation, we employ an in-cache buffer.

Buffering of writes also helps group read and write operations. Memory subsystems may have a turnaround time between read and write operations [15]. In addition, grouping of accesses to each array improves memory performance by allowing more contiguous access.

Note that buffering for writes increases the cache pressure for the L1 cache. The choice is architecture-dependent and is decided empirically at installation time.

3.5.2 Loop Order Selection

The choice of loop order determines which array is accessed contiguously across the processing of different tiles. This can result in reading or writing of contiguous cache lines. Different memory subsystems are biased towards either contiguous reads or contiguous writes. For example, on the Pentium 4, read of a cache line results in the prefetching of the subsequent cache line. This benefits biasing the loop order to processing contiguous source tiles.

The choice of loop order also interacts with the tile sizes chosen, as was described in Section 3.4. It is non-trivial to determine the appropriate loop order analytically. We empirically determine the appropriate loop order on a given machine at installation time. Note that the appropriate choice depends on the target architecture and is independent of the problem instance.
3.6 Exploiting SIMD instruction sets

The matrix transposition within the innermost tile is referred to as the micro-matrix transposition. On a given architecture, there are often many instruction sequences to do micro-matrix transposition. This is due to the various choices available in the SIMD instruction sets in doing the permutation operation. In addition, the order of instructions and the choice of memory access operations makes a difference. Less critical as most modern architectures support out-of-order execution. The different versions using the SIMD instruction sets correspond to different procedures to effect the transposition, and hence have different memory access patterns, instruction sequences, and register usage.

Data alignment constraints are imposed by most SIMD extensions. It implies that a memory access should be at an address divisible by the vector length. As one exception, SSE supports unaligned loads/stores. We need to obtain different versions of micro-transposition for the cases with and without perfect vector alignments.

Even if the base address of an array is aligned along vector boundaries, when dimensions are not multiples of the vector size, the vectors involved in a micro-transposition are not aligned for vector loads or stores.

Following the shifting approach [40], we can shift a data stream to handle vector misalignments. Therefore $n + 1$ aligned memory accesses and $n$ data shifting operations are needed to obtain $n$ contiguous vectors in a stream. In our case, $V_e$ rows in a data tile are treated as $V_e$ data streams. If the intra-tile loop order for the innermost tile is selected to be $(i,j)$, $\text{tilesize}_2/V_e + 1$ loads are needed to read each row in the source tile. In this case, if the destination tile is misaligned, we cannot write it with the same strategy because $\text{tilesize}_1$ registers are needed to use the shifting procedure efficiently. When loop order
(j,i) is selected, we have an opposite result that prefers vector writes but not vector reads. We assume the (i,j) order in the discussion below.

With AltiVec, it is simple to conduct the shifting procedure because the permutation pattern can be computed at run time using instructions \texttt{vsl} and \texttt{vsr}. Only one micro-transposition is needed for all the misaligned cases of an array. For the other array, if it is not vector aligned, scalar instructions have to be used.

While with SSE, because there is no native support for inter-register shifts, the shifting patterns must be known at compile time or misaligned memory accesses in SSE must be utilized. Given that the tile sizes are always multiples of $V_e$, it is guaranteed that every complete tile shares the same alignment pattern as shown in Figure 3.3. The alignment of a tile row is decided by both the base address $Addr_{src}$ and $N2$. For any $V_e$, $V_e^2$ alignment patterns exist. We peel the first $(Addr_{src} \mod E)$ loops to make the first row in every tile vector-aligned thus reduce the number of alignment patterns to $V_e$. For example, following this approach, when $V_e = 4$, all the 4 possible alignment patterns in a tile are (0,0,0,0,...), (0,1,2,3,...), (0,2,0,2,...), (0,3,2,1,...). For multimedia extensions such as SSE lacking flexible data reorganization support, this approach is necessary to control the number of code versions. Consequently, multiple versions of mini-transposition are necessary for SSE if misaligned memory accesses do not perform better.

### 3.7 Integrated Optimization Framework

In this section, we discuss the generation of a matrix transposition library without information on the specific problem instance. The pseudo-code for the generated code with write buffering, TLB tiling, and one level of tiling for spatial locality is shown in Figure 3.5.
Figure 3.4 depicts the procedure used to generate the library. The best micro-transposition kernels are first selected by executing each candidate on in-cache arrays. Note that more than one kernel might have to be evaluated, each corresponding to a degree of misalignment.

The architectural parameters are then used to analytically determine the set of candidate tile sizes for the two levels of tiling and the TLB tiling. Then a version of code is generated
//Notation:
//tilesize_TLB : TsizeTLB
//tilesize_1 : T1
//tilesize_2 : T2
//in-cache buffer : BUF[T1 * T2]
//
1) for iTLB = 0 to N1-1 step tsizeTLB
2) for jTLB = 0 to N2-1 step tsizeTLB
3) for iT = iTLB to iTLB+tsizeTLB-1 step T1
4) for jT = jTLB to jTLB+tsizeTLB-1 step T2
5)  // Mini-transposition
6) for i = iT to iT+T1-1 step Ve
7) for j = jT to jT+T2-1 step Ve
8)  // Micro-transposition
9) Load Ve vectors A[i][j],...,A[i+Ve-1][j+Ve-1]
10) In-register matrix transposition
11) Store these Ve vectors to BUF
12) for j = jT to jT+T2-1
13) Load A[iT][j],A[iT+1][j],...,A[iT+T1-1][j] from BUF
    as T1/Ve vectors
14) Store these vectors using non-temporal writes from BUF to
    B[j][iT],B[j][iT+1],...,B[j][iT+T1-1]

Figure 3.5: Matrix transposition code with one-level tiling and buffering writes

for each degree of misalignment using the different micro-transpose kernels. Different vari-
ations of these versions are generated for all the optimization parameters to be determined
at installation time. Variations corresponding to illegal parameter combinations are pruned
away. For example, with misaligned inputs, using SIMD instructions on AltiVec requires
buffering on at least one array due to the lack of misaligned memory accesses. So it is not
feasible to have a variation without buffering while using SIMD instruction set.

The different variations of each version are then empirically evaluated and the variation
achieving the best average performance, in terms of bandwidth, for each version is chosen.
Figure 3.6: Runtime decision tree for implementation selection. The leaves of the tree correspond to the implementations

For each version, two implementations are generated, one with buffering for reads, to handle inputs with conflict misses, and another without. All the implementations thus generated comprise the matrix transposition library.

The runtime decision tree is shown in Figure 3.6. The presence of conflict miss is first verified either analytically or using the inspector described earlier. This is used to choose between the implementations with or without read buffering. Then the appropriate version of code is chosen depending on the alignment of the arrays with respect to vector size.
3.8 Experimental Results

We evaluate the performance of the library generated on the Intel Pentium 4 and PowerPC G5. The code generation parameters that were determined analytically are shown in Table 3.3. Note that on both machines the L1 cache is large enough to hold two $L_{2e} \times L_{2e}$ tiles and $L_1 = L_2$, thus allowing the innermost tile size to be $L_{2e}$. Also note that there is no TLB tiling on Pentium 4 because $\text{tilesize}_{\text{TLB}} = \text{tilesize}_{1,2}$.

For the cache-aligned case, 8 legal parameter combinations were evaluated for Pentium 4 and PowerPC G5. For the misaligned cases, 8 legal combinations were evaluated for Pentium 4, but only 6 combinations on PowerPC G5. 2 combinations with (i,j) order but without copying for writes are not possible because there is no misaligned memory access on PowerPC. The empirically determined parameters are shown in Table 3.4. Note that the presence or absence of write buffering does not affect the performance on Pentium 4 for cache-aligned inputs. The installation procedure took several minutes on both machines.

In Figure 3.7(a) and Figure 3.7(b), we demonstrate the contributions of individual optimizations to the overall improvement in performance. Since our optimization procedure is not incremental in nature, we chose an arbitrary order of applying the optimizations. For each set of optimization parameters the best version is chosen and is evaluated on a large number of randomly selected array sizes. With the same group of optimizations, the average performance of the best version over a large number of data points is shown. The evaluation of both the scalar and vector implementations is presented. The performance of the compiler generated code is presented in the captions to the figures. The effectiveness of memory hierarchy management is demonstrated by the performance improvement obtained in matrix transposition using only scalar instructions over the compiler-generated code. For
example, *SIMD:2-level tiling + copy* denotes that we apply two-level cache tiling and include buffering for writes as an option but do not apply TLB tiling. We can observe that the SIMD version outperforms the scalar version with same optimization considerations. For example, on Pentium 4, the best SSE code has a speedup of 2.24 over the best scalar version. Several versions are not shown because some combinations of parameters are not legal or not considered with aligned cases. For example, we have to use copy to handle misaligned inputs on G5 therefore tiling without copying for writes can not be applied.

![Table 3.3: Analytically-determined parameters for single-precision matrix transposition](image)

<table>
<thead>
<tr>
<th></th>
<th>Pentium 4</th>
<th>PowerPC G5</th>
</tr>
</thead>
<tbody>
<tr>
<td>tilesize&lt;sub&gt;1&lt;/sub&gt;</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>tilesize&lt;sub&gt;2&lt;/sub&gt;</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>second-level tile size</td>
<td>80</td>
<td>160</td>
</tr>
<tr>
<td>TLB tile size</td>
<td>16</td>
<td>480</td>
</tr>
</tbody>
</table>

Table 3.3: Analytically-determined parameters for single-precision matrix transposition

![Table 3.4: Empirically determined parameters for single-precision matrix transposition](image)

<table>
<thead>
<tr>
<th></th>
<th>Misaligned input</th>
<th>Aligned input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop order</td>
<td>Pentium 4</td>
<td>PowerPC G5</td>
</tr>
<tr>
<td></td>
<td>(j,i)(iT,jT)</td>
<td>(j,i)(iT,jT)</td>
</tr>
<tr>
<td>Write buffering</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Loop order</td>
<td>(j,i)(iT,jT)</td>
<td>(j,i)(iT,jT)</td>
</tr>
<tr>
<td>Write buffering</td>
<td>either</td>
<td>either</td>
</tr>
</tbody>
</table>

Table 3.4: Empirically determined parameters for single-precision matrix transposition

We observe that in most cases adding optimization components such as whether to copy for writes or to use an additional level of tiling improve performance, which shows the benefit of the additional optimization components. The only exception is with misaligned...
inputs, TLB tiling degrades performance after two-level cache tiling is applied. We are investigating the cause and hope to incorporate this consideration into our future work.

Figure 3.7: Effect of adding optimization considerations on (a) Pentium 4 and (b) PowerPC G5.
Figure 3.8(a) and Figure 3.8(b) show the performance of different versions when $N_1 = N_2 = N$ on Pentium 4 and PowerPC G5, respectively. Inputs are cache-aligned and all the optimization parameters are considered. Not all the versions are shown here due to space limitations, but the best version is always shown. \texttt{memcpy} is the performance of the function \texttt{memcpy} provided by the operating system. Our best version performs close to the \texttt{memcpy} provided on Pentium 4 because \texttt{memcpy} is implemented in the RedHat Enterprise Linux AS 3 release as a scalar copy. On the other hand, Mac OS X provided a highly tuned SIMD implementation of \texttt{memcpy} as part of its kernel thus it outperforms our best version by about a factor of 2.

In our experiments on Pentium 4, we have $L_1$ and $L_2$ reported as 64 bytes by PerfSuite [69], which gathers hardware information of Intel processors. However, 128 bytes is often reported by micro-benchmarks as Pentium 4’s L2 cache line size such as in [123]. This is because a BIOS feature called “adjacent sector prefetch” is enabled to prefetch the next cache line for a read. On our experimental platform, this feature is also enabled as on most desktop Pentium 4 systems. This explains why the version with (iT,jT) order outperforms the version with (jT,iT) order by 43% since the latter version does not use the prefetched cache line before it is evicted. By employing an empirical search strategy, we implicitly exploit this architectural feature without knowing its existence.

On PowerPC G5, the two best versions offer almost the same performance and the only difference between them is whether copying is used for writes. In Figure 3.7(b), we can also observe that any two versions, with or without copying for writes as an option, have the same average performance since adding copying only slightly degrades the performance. However, on Intel Pentium 4 the best version with buffer copy significantly outperforms the version without copying for writes. The reason is possibly that G5 has a dedicated load
Figure 3.8: Performance of different versions with cache-aligned arrays on (a) Pentium 4 and (b) PowerPC G5
bus and a dedicated store bus and both buses work in a fully independent fashion. Also, the cacheability control on G5 is not as efficient as non-temporal writes on Pentium 4.

The (j,i) intra-tile order always performs better than (i,j) order on both platforms. We believe with (j,i) order there is better exploitation of DRAM’s pipelined parallelism because of the increased possibility of accessing different memory banks simultaneously.

Many architectural features such as memory-bus design or DRAM types vary greatly with architectures and even with specific machine configurations. The exploitation of these architecture-specific features are often out of the scope of most model-driven compiler optimizations. The above observations demonstrate the effectiveness of our empirical search in exploiting these “hidden” architectural features, especially for memory bandwidth-bound programs.

Figure 3.9(a) and Figure 3.9(b) demonstrate the performance of transposing $N \times N$ misaligned single-precision matrices on Pentium 4 and PowerPC G5, respectively. In the presence of cache misalignments, different parameter combinations are selected as the best as compared to the cache aligned case. Unlike the cache aligned case, the variation in the performance of different versions is not significantly different on Pentium 4. The version without buffering for writes outperforms the other versions. The misalignment precludes the use of non-temporal writes and complicates the memory bus traffic pattern. This possibly leads to the benefits from buffering for writes being overshadowed by the buffering overhead. On PowerPC G5, the $(jT,iT)(j,i)$ combination is selected as the best version. In comparison, it underperforms the $(iT,jT)(j,i)$ combination with cache-aligned inputs.
Figure 3.9: Performance of different versions with cache-misaligned arrays on (a) Pentium 4 and (b) PowerPC G5
3.9 Extending to High Dimensions

We extend the approach presented in previous sections in order to optimize arbitrary index permutations of multi-dimensional arrays, the generalized form of matrix transposition. Index permutation is a key operation in many scientific applications such as those in computational chemistry, weather modeling, computational oceanology, and data layout transformation to improve locality in programs [61, 57]. When going to high dimensions from matrix transposition, there exist three additional challenges and we answer them with changes to our original tuning procedure of matrix transposition.

1. It is important to reduce the number of generated code versions while optimizing for different permutations. Instead of having $n!$ code versions for all possible permutations of $n$-dimensional arrays, we only generate $n$ versions. That is, by always accessing the source array or the destination array in a certain order, we are able to calculate the access stride of each loop to the other array. In such a way one code version handles $(n-1)!$ permutations.

2. The decrease in dimension sizes with the increasing dimensionality impairs the benefits from optimizations such as loop tiling. Due to the limited benefit of second-level tiling and TLB tiling with reduced dimensions, we only have one level of tiling when dimensionality $n$ is larger than 3.

3. The index calculation overhead must be effectively controlled to achieve high performance. Instead of relying on compiler-generated code, we identify loop invariants and generate efficient indexing code by strength reduction [5].

Following the optimization procedure with the above changes, we have been able to obtain a highly optimized index permutation library on both PowerPC G5 and Pentium
4. Instead of presenting absolute bandwidth numbers, we demonstrate the effectiveness of optimized index permutation operations by employing them in NWChem [3], a widely used computational chemistry package developed by the Pacific Northwest National Laboratory.

In the quantum chemistry programs in NWChem 4.7, there are two variants of index permutation operations used. One version is the same as what we have used in Chapter 2. It can be represented as $A' = \text{permute}(A, p)$, where the layouts of $A'$ and $A$ must conform after permutation $p$. The usage of this operation is to transform array layouts in order to use $GEMM$ in the $BLAS$ library, same as in Chapter 2. The other variant is represented as $A' = A' + c \times \text{permute}(A, p)$, where $c$ is a constant and the layouts of $A'$ and $A$ must conform after permutation $p$. It is mainly used in the so-called symmetrization operations where arrays with certain symmetric properties are constructed from asymmetric arrays.

Two important quantum chemistry computations are used in our evaluation: (1) triples correction in the $CCSD(T)$ computation and (2) the $CCSDT$ computation. The experiments with NWChem 4.7 were conducted on the same Pentium 4 platform used in previous sections. By replacing the original index permutation code in NWChem with ours, we have been able to obtain significant performance improvements with different inputs, as shown in Figures 3.10 and 3.11. In particular, we have the following observations.

1. The computation complexity of triples correction is $O(\Theta^3 \Theta^4)$ while the index permutation cost of triples correction is $O(\Theta^3 \Theta^3)$ which is mainly on symmetrization operations. It is known that index permutation dominates the computation of triples correction. Our implementation offers overall speedups of 2.27 and 2.58 for two different molecules. This improvement essentially comes from the index permutation speedups of 3.35 and 3.53.
2. The computation complexity of CCSDT is $O(O^3V^5)$ while its index permutation cost is only $O(O^3V^3)$. It does not seem that optimizing index permutation can significantly improve the overall performance. Interestingly we are able to achieve overall speedups of 2.02 and 1.74 for two large inputs. This is because many computations are for small molecules with large basis sets and index permutation operations can still have a high cost in these cases due to the difficulties in optimizing them.

![Normalized execution time of triples correction with different molecule inputs on Pentium 4.](image)

**Figure 3.10:** Normalized execution time of triples correction with different molecule inputs on Pentium 4.

### 3.10 Related Work

Theoretical study and empirical evaluation of optimizing matrix transposition with cache performance considerations was conducted in [21, 44]. The authors conclude that, assuming conflict misses are unavoidable, it is impossible to be both cache efficient and register efficient, and employ an in-cache buffer. Other memory characteristics are not
taken into account. Zhang et al. [126] focus on how to write an efficient bit-reversal program with loop tiling and data padding. We do not have data padding as an option since we focus on generation of a library that cannot change layout of its inputs. The primary focus of that paper is on conflict misses, which is important in bit-reversal. We showed in Section 3.2 that conflict misses are not as widespread in the case of matrix transposition.

Different implementations of matrix transposition are investigated by Chatterjee et al. [25], with the conclusion that hierarchical non-linear layouts are inherently superior to the standard layouts for the matrix transposition problem. Optimizing for such layouts is beyond the scope of this dissertation.

There have been studies on how to achieve space-efficiency in matrix transposition or its more generalized forms [12, 70, 14, 37]. Our present work does not handle in-place transposition. We intend to handle in-place transposition by carefully ordering the transposition at the tile level and marking transposed tiles.
Several studies focus on how to generate or optimize intra-register permutations. The generation of register-level permutations is addressed in [68]. The algorithm optimizes data permutations at the instruction level and focuses on SSE instructions. Ren et al. [100] present an optimization framework to eliminate and merge SIMD data permutation operations with a high-level abstraction. Both studies propagate data organization along data-flow graphs and focus on reducing intra-register permutations. We manually generate various versions of micro-kernels and empirically choose the best one. While limiting, the manual process is only repeated once for every vector instruction set. The limited number of vector instruction sets allows this process to be applicable across a wide range of processor architectures.

Empirical search employed in library generators such as ATLAS [118, 117, 122] has drawn great interest because of the complexity of analytical modeling of optimal parameters for modern architectures. However, empirical global search is often too expensive to apply. Yotov et al. [124] present a strategy employing both model-driven analysis and empirical search to decide optimization parameters in matrix multiplication. Chen et al. [29] also present an approach to combining compiler models and empirical search, using matrix multiplication and Jacobi relaxation as two examples. Our work is similar in spirit but is applied to a computation that is bandwidth-limited and has no temporal locality. Matrix transposition is similar to the level 1 BLAS kernels optimized by Whaley and Whalley [119] using an empirical search-based approach. But the presence of strided memory access in matrix transposition makes it harder to exploit spatial locality.
3.11 Summary

Extensive research has been conducted on optimizing index permutation and related problems because of their ubiquitous usage and unique memory access patterns. In this chapter, we presented our approach employing both offline analysis and empirical search to decide optimization parameters for index permutation. We handle various alignments and conflict misses by generating multiple versions. Significant improvements are reported on Intel Pentium 4 and PowerPC G5 platforms with code generated by a special code generator. Several interesting observations demonstrate the effectiveness of our approach in exploiting hard-to-optimize architecture-specific features in the cache hierarchy and the memory subsystem. We believe our approach is very promising in optimizing other memory bandwidth-bound kernels and streaming applications.
In this section, we present how we compile data-parallel programs using non-canonical data layouts for future chip multiprocessors with a Non-Uniform Cache Architecture (NUCA).

4.1 Introduction

Many proposed chip multiprocessor (CMP) designs [113, 56, 66] feature shared on-chip cache(s) to reduce the number of off-chip accesses and simplify coherence protocols. With diminutive feature sizes making wire delay a critical bottleneck in achieving high performance, proposed shared caches employ a Non-Uniform Cache Architecture (NUCA) design that spreads data across cache banks that are connected through an on-chip network [64, 51, 11]. Figure 4.1(a) shows a tiled architecture for a CMP. Each tile contains a processor core with private L1 cache, and one bank of the shared L2 cache. Future CMPs are likely to be based on a similar design to enable large numbers of processors and large on-chip caches to be connected through emerging technologies such as 3D stacking. Figure 4.1(b) shows the mapping of a physical memory address to banks in the L2 cache. The address space is block-cyclically mapped across the banks with a block size $L$. The lowest $\log_2 L$ bits represent the displacement within a block mapped to a bank, and the next
set of \(\log_2 P\) bits specify the bank number. The value of \(L\) depends on the granularity of interleaving across the banks, which can range from one cache line to an OS page. The Sun UltraSPARC T1 [66] uses a cache-line interleaved mapping scheme for its banked L2 cache. Cache line interleaving has a number of benefits, including uniform distribution of arrays across the banks, enabling high collective bandwidth in accessing its elements and minimization of hot-spots. Since bank interleaving at cache line granularity has been used in most NUCA CMP designs, the evaluations in this chapter assume cache-line interleaving. However the developed approach is also applicable to coarser interleaving across banks.

In this chapter, we develop a compile-time data locality optimization framework for bank-interleaved shared cache systems. We first use a simple example to illustrate the optimization issue and the proposed data layout transformation approach.

Consider the code in Figure 4.2, for 1-D Jacobi iteration. In statement S1 of the code, iteration \(i\) accesses \(A[i - 1]\), \(A[i]\), and \(A[i + 1]\). Completely localized bank access is impossible with this code for any load-balanced mapping of iterations to processors. For ease of explanation, we assume that the array size \(N\) is a large power of two. Assuming there are four processors on a tiled CMP and \(L = 2\) is the cache line size in elements, there are many ways of partitioning the iteration space to parallelize this program. Consider the following two mapping schemes with one thread on each processor: (1) based on the owner-computes rule, each thread is responsible for computing the data elements mapped to its local bank — this corresponds to OpenMP’s static scheduling with a chunk size of \(L\); (2) the iteration space is divided evenly into four contiguous partitions and they are assigned to the processors in order — this corresponds to OpenMP’s default static scheduling. Note that aligning the base addresses of \(A\) and \(B\) is important since \(A[i]\) and
B[i] are always referenced together. Assume that the origins of A and B are both aligned to bank 0’s boundary. Figure 4.3(a) shows the data communication needed between different processors when the computation is partitioned using mapping (1). In order to compute B’s elements in a cache line, the corresponding cache line of A, as well as the previous and next cache line are required. (for the A[i-1] reference for the lowest iteration in the block and A[i+1] reference for the highest iteration in the block). With two cache lines of A being remote, the total inter-bank communication volume in every outer iteration is roughly $2^N$.

For mapping (2), if we keep the original layout of each array, most data accesses by each
while (condition) {
    for (i = 1; i < N-1; i++)
    for (i = 1; i < N-1; i++)
        A[i] = B[i]; //stmt S2 }

Figure 4.2: 1D Jacobi code.

![Diagram of parallelizing a 1D Jacobi program on a 4-tile CMP](image)

Figure 4.3: Parallelizing a 1D Jacobi program on a 4-tile CMP.

processor are remote. Furthermore, such remote communications can be across the entire chip instead of only among neighboring tiles. However, it is possible to rearrange the data layout in order to significantly enhance locality of access and maintain affinity between
computation and data. This is shown in Figure 4.3(b). As illustrated, there are only six remote accesses in every outer iteration.

The basic problem illustrated by this example is a mismatch between the fundamental data affinity relationships inherent to the algorithm and the locality characteristics imposed by the hardware bank-mapping. The nature of the Jacobi computation is that each data element of the 1D array $A$ has affinity with its immediate neighbors, since they are operands in common arithmetic operations. Thus data locality can be optimized by grouping a set of contiguous elements of $A$ on a processor, along with mapping the operations that access the elements. However, this is at odds with the block-cyclic bank mapping imposed by the hardware which maps successive cache lines to distinct banks in order to minimize hotspots. The solution is to perform a suitable data layout transformation so that data elements with strong affinity to each other get mapped to the same processor’s bank under the imposed hardware mapping scheme.

The chapter develops a general framework for integrated data layout transformation and loop transformations for enhancing data locality on NUCA CMP’s. While the direct focus of the chapter’s presentation is to detail the compilation framework for this target, the basic approach has much broader applicability that is the subject of ongoing work - effective vectorization for short SIMD architectures, domain-specific language and compiler for stencil computations, and a compiler for linear algebra with recursive arrays [7].

Our framework uses the polyhedral model of compilation. The polyhedral model provides a powerful abstraction to reason about transformations on collections of loop nests by viewing a dynamic instance (iteration) of each statement as an integer point in a well-defined space called the statement’s *polyhedron*. With the polyhedral model, it is possible to reason about the correctness of complex loop transformations using powerful machinery
from linear algebra and integer linear programming. A significant advantage of using a polyhedral compiler framework is that imperfectly nested loops are very cleanly handled as are compositions of loop and data transforms. Although the foundational ideas for polyhedral compilation were developed in the early nineties [42, 41, 93, 62], it was generally considered to be too expensive for practical use. However a number of recent developments [94, 34, 92, 13] have demonstrated the practical effectiveness of the approach. Now at least two commercial compilers are known to incorporate polyhedral analyses; the latest release of GCC [45] also includes it.

The approach we develop has two main steps. First, an affinity characterization of the data arrays is carried out using a “localization” analysis. In this step an affine mapping of iteration spaces and data spaces onto a logical 1D target space is sought such that no iteration accesses any data that is beyond a bounded distance in the target space. If such an affine mapping exists, our analysis is guaranteed to find it. The existence of such a mapping implies that the computation is localizable, i.e. data affinities among the elements of arrays are constrained to be close by along some dimension of the array. Consequently, a mapping of iterations to processors along with a suitably transformed data layout is feasible that will result in enhanced data locality.

The localization analysis identifies dimensions of data arrays that are to be layout-transformed. A key issue is that of efficient code generation that indexes into the transformed arrays. This is done by using CLooG. A straightforward use of CLooG results in very inefficient code with significant indexing overhead in the innermost loops. Efficient code is generated by differentiating the “bulk” scenario from the “boundary” case via linear constraints; the latter region is traversed without expensive indexing calculations, which are only needed for the smaller boundary regions.
The chapter makes the following contributions:

- It develops an approach for automatic data layout transformation to enhance data locality on emerging NUCA architectures.
- It develops an effective code generation framework that can be adapted for an important class of data layout transformations, including those to enhance vectorization of stencil codes.

The rest of the chapter is organized as follows. We first present the framework that characterizes data affinity via data localization analysis in Section 4.2. In Section 4.3, we describe the affinity characterization is used to determine data layout transformations. The key challenge of efficient code generation to access the layout-transformed data is addressed in Section 4.4. In Section 4.5, using several benchmarks we evaluate the effectiveness of our approach with an architecture simulator. We discuss related work in Section 4.6 and present our conclusions in Section 4.7.

4.2 Localization Analysis

This section describes the procedure to determine the localizability of a given program. The analysis is based on a polyhedral transformation framework. We begin by providing some background, using notation similar to that used by Griebl [48].

4.2.1 Polyhedral Framework for Program Transformation

A $d$-dimensional function $f$ of $n$ variables $x_1, x_2, \ldots, x_n$ is affine if and only if $f(\vec{x}) = M_f \vec{x} + c$, where $\vec{x} = \begin{bmatrix} x_1 & \ldots & x_n \end{bmatrix}^T$, $M_f \in \mathbb{R}^{d \times n}$ and $c \in \mathbb{R}^{d \times 1}$. An affine function $f$ can be expressed in a linear form by introducing a special constant parameter “1”: $f(\vec{x}') = M_f' \vec{x}'$, where $\vec{x}' = \begin{bmatrix} x_1 & \ldots & x_n & 1 \end{bmatrix}^T$ and $M_f' = [M_f \ c]$.  

75
A hyperplane is a \((d - 1)\)-dimensional affine subspace of a \(d\)-dimensional space and can be represented by an affine equality \(Mx + c = 0\). A halfspace consists of all points of the \(d\)-dimensional space which are on one side of a hyperplane, including the hyperplane. A halfspace can be represented by an affine inequality \(Mx + c \geq 0\). A polyhedron is the intersection of a finite number of halfspaces. A polytope is a bounded polyhedron.

The bounds of the loops surrounding a statement \(S\) are expressed as a system of linear inequalities that form a polytope, represented as \(D_S(\vec{i}) \geq 0\), where \(\vec{i}\) includes the iteration vector and all structure parameters. An array access in statement \(S\) to array \(A\) is denoted \(F_{S,A}(\vec{i})\). Data dependences are described using \(h\)-transformations. An \(h\)-transformation is an affine function that takes a destination statement instance and gives the source statement instance \([42]\). Note that we are using value-based dependences. The domain of the function is denoted explicitly as a polytope \(P_e(\vec{i}) \geq 0\).

We are interested in a computation allocation \(\pi_S\) that is an affine function mapping every dynamic instance of a statement \(S\) to an integer vector that represents a virtual processor. \(\pi_S\) must satisfy the constraints imposed by loop bounds and data dependences. Similar to computation allocation, a data mapping \(\psi_A\) maps every array element to a virtual processor.

The problems of finding \(\pi_S\) and \(\psi_A\) are often formalized as optimization problems. However, such optimization problems are not affine because coefficients in \(\pi_S\) or \(\psi_A\) and the iteration vector are all unknowns. Farkas Lemma \([42]\) is used in such cases for quantifier elimination. This lemma states how such a system of affine inequalities with quantifiers can be transformed to a system of affine equalities by adding non-negative variables, referred to as Farkas multipliers. After such transformations \(\pi_S\) and \(\psi_A\) can be determined by using a linear programming solver.
Lemma 1 (Farkas Lemma) Let $\mathcal{D}$ be a non-empty polyhedron defined by $p$ inequalities $a_j x + b_j \geq 0$, $1 \leq j \leq p$. Then, an affine form $f$ is non-negative everywhere in $\mathcal{D}$ if and only if it is a positive affine combination:

$$f \equiv \lambda_0 + \sum_{j=1}^{p} \lambda_j(a_j x + b_j), \quad \lambda_j \geq 0 \land 0 \leq j \leq p. \quad (4.1)$$

The non-negative constants $\lambda_j$ are referred to as Farkas multipliers.

4.2.2 Localization Constraints

We define a program as *localizable* if there exists a computation mapping and a data mapping to a target 1-dimensional space such that the data required by any iteration instance is mapped in a bounded neighborhood in the target space. The existence of such a mapping enables suitable layout transformation to enhance locality and reduce the total amount of non-local data accesses. The localizability constraints for computation allocation are formalized as follows.

**Definition 4.1 (Localized computation allocation and data mapping)** For a program $P$, let $D$ be its index set, computation allocation $\pi$ and data mapping $\psi$ for $P$ are localized if and only if for any array $A$, and any reference $F_{S,A}$, $\forall \vec{i}, \ D_S(\vec{i}) \geq 0 \implies |\pi_S(\vec{i}) - \psi_A(F_{S,A}(\vec{i}))| \leq q$, where $q$ is a constant.

As a special case, *communication-free localization* can be achieved if and only if for any array $A$ and any array reference $F_{S,A}$ in a statement $S$, computation allocation $\pi$ and data mapping $\psi$ satisfy $\psi_A(F_{S,A}(\vec{i})) = \pi_S(\vec{i})$.

4.2.3 Localization Analysis Algorithm

The localization analysis based on the above discussion consists of the following steps, as described in Algorithm 4.1.
**Step 1: Grouping Interrelated Statements/Arrays** We determine connected sets of statements in an input program. We form a bipartite graph where each vertex corresponds to a statement or an array, and edges connect each statement vertex to all arrays referenced in that statement. We then find the connected components in the bipartite graph. The statements in each connected component form an equivalence class.

**Step 2: Rewriting Array Indices** We rewrite the program such that all array references are to byte arrays. For example, a reference $A[i][j + 1]$ to a double array $A$ is rewritten as $A_{\text{byte}}[i][8j + 8]$. This allows us to account for different data sizes since the default layout maps data based on their byte addresses. This is explained in detail in Section 4.4.

**Step 3: Finding Localized Computation Mapping** Following Def. 4.1, we formulate the problem as finding an affine computation allocation $\pi$ and an affine data mapping $\psi$ that satisfy $|\pi_S(\vec{i}) - \psi_A(F_{S,A}(\vec{i}))| \leq q$ for every array reference $F_{S,A}$. The $\pi$ function identifies a parallel space dimension in the iteration space of each statement. Therefore a system of such inequalities is formed as constraints for all statements and array references.

Note that the equation in Def. 4.1 is not affine due the the quantifier. We need to first rewrite each constraint as

\[
\forall \vec{i}, \; D_S(\vec{i}) \geq 0 \implies \pi_S(\vec{i}) - \psi_A(F_{S,A}(\vec{i})) + q \geq 0; \quad (4.2)
\]

\[
\forall \vec{i}, \; D_S(\vec{i}) \geq 0 \implies -\pi_S(\vec{i}) + \psi_A(F_{S,A}(\vec{i})) + q \geq 0. \quad (4.3)
\]

For simplicity of presentation, we refer to $\pi_S(\vec{i}) - \psi_A(F_{S,A}(\vec{i})) + q$ and $-\pi_S(\vec{i}) + \psi_A(F_{S,A}(\vec{i})) + q$ as $f_1(\vec{i})$ and $f_2(\vec{i})$ respectively.

We apply Farkas Lemma to transform the above constraints to affine equalities by introducing Farkas multipliers. Take the first equation in (4.2) as an example. With Farkas Lemma, we have $\forall \vec{i}, \; D_S(\vec{i}) \geq 0 \implies f_1(\vec{i}) \geq 0$ if and only if $f_1(\vec{i}) \equiv \lambda_0 + \sum_k \lambda_k (a_k \vec{i} + b_k)$
with $\lambda_k \geq 0$, where affine inequalities $a_k\vec{i} + b_k \geq 0$ define $D_S$, and $\lambda_k$ are the Farkas multipliers. Therefore we have the following:

$$M'_{f_1} \begin{bmatrix} \vec{i} \\ 1 \end{bmatrix} = \begin{bmatrix} \lambda_1 & \ldots & \lambda_m & \lambda_0 \end{bmatrix} \begin{bmatrix} M'_{D_S} \\ 0 \ldots 01 \end{bmatrix} \begin{bmatrix} \vec{i} \\ 1 \end{bmatrix}, \vec{\lambda} \geq 0 \tag{4.4}$$

where $f_1(\vec{i}) = M'_{f_1} \begin{bmatrix} \vec{i} \\ 1 \end{bmatrix}$ and $D_S(\vec{i}) = M'_{D_S} \begin{bmatrix} \vec{i} \\ 1 \end{bmatrix}$. Since Eq. (4.4) holds for all $\vec{i}$,

$$M'_{f_1} = \begin{bmatrix} \lambda_1 & \ldots & \lambda_m & \lambda_0 \end{bmatrix} \begin{bmatrix} M'_{D_S} \\ 0 \ldots 01 \end{bmatrix}, \vec{\lambda} \geq 0 \tag{4.5}$$

We apply Fourier-Motzkin elimination to Eq. (4.5) to remove all the Farkas multipliers and obtain a new set of inequalities. The whole procedure is also applied to the second equation in (4.3) to obtain another set of inequalities. These two sets together form the inequalities that constrain the coefficients of $\pi$ and $\psi$.

In order to minimize potential communication, the system of generated inequalities is solved as an integer programming problem. $\min(q)$ is used as the objective function to minimize the communication overhead. If such a $q$ is identified, we find localized computation allocation and data mapping. As a special case, if $q$ is determined to be 0, the computation is identified to have communication-free computation localization. If a non-zero $q$ is determined as the minimal solution, the program is not communication-free but is localizable. If a solution is not obtained by the above steps, the program is identified as non-localizable.

**Partial Localization** We also consider partial localization. That is, we allow a processor to have a small number of data accesses outside its neighborhood. If localized $\pi$ and $\psi$ cannot be found, we attempt to find partial localization by removing constraints on small arrays with high reuse. Then we recompute $\pi$ and $\psi$, seeking localization with the reduced constraints — localization of small arrays being ignored in the expectation that temporal locality may be exploited via registers or in the private L1 caches.
Algorithm 4.1 Localization analysis algorithm

Require: Array access functions after indices are rewritten to access byte arrays

1: \( C = \emptyset \)
2: for each array reference \( F_{S,A} \) do
3: \begin{align*}
& \text{Obtain new constraints: } \pi_S(\vec{i}) - \psi_A(F_{S,A}(\vec{i})) + q \geq 0 \text{ and } \psi_A(F_{S,A}(\vec{i})) - \pi_S(\vec{i}) + q \geq 0 \\
& \text{under } \vec{i} \in D_S.
\end{align*}
4: Apply Farkas Lemma to new constraints to obtain linear constraints; eliminate all Farkas multipliers
5: Add linear inequalities from the previous step into \( C \)
6: Add objective function (\( \min q \))
7: Solve the resulting linear programming problem with constraints in \( C \)
8: if \( \psi \) and \( \pi \) are found then
9: \quad return \( \pi, \psi, \) and \( q \)
10: return “not localizable”

for \( i = 0; \ i < N; \ i++ \) {
    C[i] = 0; //stmt 1

    for \( j = 0; \ j < N; \ j++ \)
}

Figure 4.4: Matrix-vector multiplication code.

As an example of partial localization, code to compute matrix-vector multiplication is shown in Figure 4.4. There are two sets of data dependences between instances \((i, j)\) and \((i', j')\) of statement 2: (1) output dependence on array \( C \): \( i - i' = 0, j - j' \geq 1 \), and (2) input dependence on array \( B \): \( i - i' \geq 1, j - j' = 0 \). No localizable mapping exists for all three arrays when considering both dependences. To overcome this problem we remove constraints on array \( B \) because \( B \)’s reuse is asymptotically higher than \( A \)’s and its size is asymptotically smaller. We are then able to obtain \( \pi_1(i, j) = i, \pi_2(i, j) = i, \psi_A(i, j) = i \) and \( \psi_C(i) = i \) that are communication-free regarding arrays \( A \) and \( C \).
4.3 Data Layout Transformation

In this section, we develop the approach to data layout transformation for arrays identified by localization analysis framework.

As discussed in the previous section, in our implementation the linear programming solver finds a lexicographically minimum solution. We consider data mappings of the form \( \psi_A(\vec{d}) = s_A \ d_k + \text{off}_A \), where \( s_A \) and \( \text{off}_A \) are small integers, specifying A’s distribution stride and offset along its \( k \)th dimension. While our localization analysis framework is general and our code generation framework can be extended to handle more complicated programs such as those having skewed data mapping \( \psi(d_1, d_2) = d_1 + d_2 \), in the rest of the chapter we focus on the above common cases. For these programs, each array not removed by partial localization has exactly one data dimension that corresponds to a surrounding parallel loop in the generated code. When such a program is determined to be localizable, we decide each array’s data layout to minimize on-chip communication as follows.

In a connected component of a bipartite graph representing a program, if we find communication-free localization where all arrays share the same stride along their fastest varying dimensions, we use the canonical data layout for each array as the computation can be arranged such that each processor core only accesses local data. Otherwise we employ non-canonical data layouts to avoid on-chip communication based on the following analysis. If a program is determined to be localizable but not communication-free, we obtain a small positive \( q \) that indicates the maximum difference in the target space, of the affine mappings of any iteration and any data element that it accesses. This means that the maximum distance of separation is \( 2 \ast s \) along \( d_k \) for any pair if array elements that are accessed by any common statement iteration instance. This implies that by partitioning the array elements
into $P$ contiguous groups along $d_k$ and by similarly grouping the corresponding accessor iterations, excellent data locality could be achieved if the array were block-distributed along $d_k$ among the $P$ processors. But the NUCA architecture uses a round-robin bank-mapping of cache lines. Since consecutive cache lines are mapped to different banks, in order to have two cache lines of consecutive data mapped to the same bank, we need to keep them at a distance of $PL$ in virtual memory, where $L$ is the cache line size and $P$ is the number of processor cores. The 1D Jacobi code in Figure 4.2 demonstrates such an example. The program is localizable but not communication-free with $q = 1$, $\pi(i) = i$ and $\psi(i) = i$.

As shown in Figure 4.3, the non-canonical data layout mentioned above eliminates most on-chip communication. When the program is communication-free but array mappings do not shared the same stride, we can follow the same approach to avoiding on-chip communication with the non-canonical layout.

The non-canonical layout discussed above can be viewed as the combination of a set of data layout transformations, including strip-mining, permutation and padding. Similar layout transformations have been used by previous studies [8, 101, 102, 6] to avoid conflict misses or reduce false sharing. However, we are unaware of any prior work that has developed a general source-to-source code transformation algorithm to automatically convert the original code addressing canonical arrays into efficient code indexing layout-transformed arrays. In the next section, we develop such a framework.

**Strip-mining** Strip-mining is analogous to loop tiling in the iteration space. After strip-mining, an array dimension $N$ is divided into two virtual dimensions ($\lceil \frac{N}{d} \rceil$, $d$) and an array reference $[...][i][...]$ becomes $[...][\frac{i}{d}][i \mod d][...]$. Note that strip-mining does not change an array’s actual layout but instead creates a different logical view of the array with increased dimensionality. As mentioned above, on a tiled CMP, a canonical view
does not provide any information about an array element’s placement. By strip-mining the
fastest varying dimension we have an additional dimension representing data placement.
For example, assuming that a one-dimensional array \( A(N) \) with canonical layout is aligned
to the boundary of bank 0, after strip-mining this array twice, there is a three-dimensional
array \( A'([\frac{N}{P}\cdot L], P, L) \) where \( L \) is the L2 cache line size and \( P \) is the number of tiles. With
this three-dimensional view, we can decide an array element’s home tile by examining its
second fastest varying dimension.

**Permutation**  Array permutation is analogous to loop permutation in the iteration
space. After permutation, an array \( A(\ldots, N_1, \ldots, N_2, \ldots) \) is transformed to \( A'(\ldots, N_2, \ldots, N_1, \ldots) \)
and an array reference \( A[\ldots][i_1][\ldots][i_2][\ldots] \) becomes \( A'[\ldots][i_2][\ldots][i_1][\ldots] \). Combined with
strip-mining, permutation changes an array’s actual layout in the memory space. Figure 4.5 shows how one-dimensional array \( A \) in the 1D Jacobi example in Figure 4.2 is
transformed with strip-mining and permutation to group consecutive data, where consecutive caches in the array are separate at a distance of \( PL \) in the memory. Figure 4.5(a)
shows the default data layout through strip-mining, where an element \( A[i] \) is mapped to tile
\( i_p = \lfloor i/L \rfloor \mod P \). In order to realize consecutive data grouping, we can first strip-mine
array \( A(N) \) so we have \( A'([\frac{N}{P}\cdot L], P, L) \) and then we have array reference \( A[i] \) rewritten as
\( A'[i_p][i_c][i_L] \). As shown in Figure 4.5(b), by permuting the first two dimensions of \( A' \), we
obtain a new array \( A'' \) and we are able to map \( A'[i_p][i_c][i_L] \) (i.e., \( A''[i_c][i_p][i_L] \)) to tile \( i_p \) such
that block distribution of \( A \) is achieved.

**Padding**  To enforce data mapping, array padding is employed in two different ways.
First, keeping the base addresses of arrays aligned to a tile specified by the data mapping
function can be viewed as *inter-array padding*. In our implementation, we use compiler
Figure 4.5: A one-dimensional example of data transformations. (a) Canonical layout after strip-mining. (b) Non-canonical layout after strip-mining and permutation.

directives such as `attribute` in GNU C to control an array’s base address. Second, *intra-array padding* aligns elements inside an array with “holes” in order to make a strip-mined dimension divisible by its sub-dimensions. In the 1D Jacobi example, if \( N \) is not a multiple of \( PL \) in Figure 4.5, we pad \( N \) to \( N' = \lceil \frac{N}{PL} \rceil \cdot P \cdot L \) and \( N' - N \) elements are unused in the computation.

In summary, we choose the data layout for each array in a localizable program and map array indices as follows.

- If we find communication-free localization where all arrays share the same stride along their fastest varying dimensions, canonical data layout is employed.
• Otherwise we follow localization analysis results to create a blocked view of n-dimensional array \( A \) along dimension \( k \). When \( k = 1 \) which is the fastest-varying dimension, necessary data layout transformations have been discussed in the above 1D Jacobi example. We have a function \( \sigma \) that maps the original array index to the index with the non-canonical layout. Here \( \sigma(i_n, i_{n-1}, \ldots, i_2, i_1) = (i_n, i_{n-1}, \ldots, i_2, i_1 \mod (N_1/P))/L, i_1/(N_1/P), i_1 \mod L) \). When \( k > 1 \), the distributed dimension and the fastest varying dimension are strip-mined to add processor and cache line offset dimensions respectively. Padding is needed if \( P \nmid N_k \) or \( L \nmid N_1 \). Finally an index permutation is applied to the processor and the second fastest-varying dimensions. We have \( \sigma(i_n, \ldots, i_k, \ldots, i_1) = (i_n, \ldots, i_1/L, i_k \mod (N_k/P)\ldots, i_2, i_k/(N_k/P), i_1 \mod L) \).

4.4 Code Generation

As discussed above, data layout transformation on arrays can be expressed as a one-to-one mapping \( \sigma \) from the current \( n \)-dimensional index domain to the transformed \((n + 2)\)-dimensional index domain. A straightforward approach to code generation would be to substitute every array reference \( A[u(i)] \) with \( A'[\sigma(u(i))] \) where \( A' \) is the transformed array. and to generate the code satisfying the given schedule by using the state-of-the-art code generation methods [94]. Unfortunately such a method would produce very inefficient programs as \( \sigma \) involves expensive expressions with integer divisions and modulos that are computed for every array reference.

The method proposed here is to iterate directly in the data space of the transformed array, so that array references are addressed directly by the iteration vector and therefore no longer involve expensive integer expressions. The usual way to generate code in the polyhedral model is to specify for each assignment an iteration domain \( D \) and an (affine)
scheduling function that gives the execution date for each iteration $i$ in $D$. Several approaches have been devoted to generate efficient code including the Quilleré-Rajopadhye-Wilde algorithm [94] which is implemented in the state-of-the-art tool CLooG [34]. The approach described hereafter follows this general methodology.

Given an assignment with uniform references $B[u_0(i)] = f(A[u_1(i)], ..., A[u_n(i)])$, whose iteration domain is $D$, one can write a new domain $D' = \{(i, j_0, ..., j_n), i \in D, j_0 = \sigma(u_0(i)), \ldots, j_n = \sigma(u_n(i))\}$, and generate the code for the schedule provided by the previous step, expressed in terms of a given $j_\ell$.

Even though $\sigma$ involves non-affine expressions with integer division and modulos, it is possible to express each constraint $j_k = \sigma(u_k(i))$ with affine constraints by using the definition of integer division. However, this method can generate a significant number of intermediate variables, leading to a code blow-up. We avoid this problem by using the fact that the inverse of $\sigma$ is (here) always affine, and to replace each constraint $j_k = \sigma(u_k(i))$ by $\sigma^{-1}(j_k) = u_k(i), j_k \in T(\sigma)$ where $T(\sigma)$ is the target domain of $\sigma$. This way, no intermediate variables will be needed to describe $D'$.

For 1-D Jacobi, original code for the first statement is: $B[i] = A[i-1] + A[i] + A[i+1]$. With data layout transformation $\sigma(i) = (i', p, l)$, the transformed code is given by: $B'[i'][p][l] = A'[i'_-][p_-][l_-] + A'[i'_+][p_+][l_+] + A'[i'_+][p_+][l_+]$ where $A'$ and $B'$ are the transformed versions of $A$ and $B$, respectively. In addition, here $\sigma(i-1) = (i'_-, p_-, l_-)$ and $\sigma(i+1) = (i'_+, p_+, l_+)$. If we compute $\sigma$, one would generate a loop nest with counters $t, p, i'$ and $l$ containing assignments defining the remaining variables in terms of $(i', p, l) (j_\ell)$, asserting that $(i'_-, p_-, l_-) = \sigma(\sigma^{-1}(i', p, l) - 1)$ and $(i'_+, p_+, l_+) = \sigma(\sigma^{-1}(i', p, l) + 1)$. These expressions
involve integer divisions and modulos and must be computed at every iteration, leading to an expensive code.

To bypass this issue, we identify an affine relation between the (transformed) array references. As we apply data layout transformations which are always simple affine expressions followed by an integer division, or a modulo by a constant, this can be achieved by using simple recipes on integer division and modulo. Indeed, when \( u < N \), one can remark that \( (i - u)/N \) is often equal to \( i/N \), and that \( (i - u) \mod N \) is often equal to \( i \mod N - u \). Applying these recipes to every \( j_k = \sigma(u_k(i)) \), we obtain a collection of affine relations \( (j_k = \alpha_k j_\ell + \beta_k, 0 \leq k \leq n) \) that characterize a class of execution instances of the assignment whose generic instance is drastically simplified, \( B'[\alpha_0.j_\ell + \beta_0] = f(A'[\alpha_1.j_\ell + \beta_1], ..., A'[\alpha_n.j_\ell + \beta_n]) \), and corresponds to a form of steady state of the computation in the iterations space.

Assume that we have computed \( \sigma(i) = (i', p, l) \) using integer division and modulo operations. Using the observation mentioned above, \( \sigma(i - 1) = (i'_-, p_-, l_-) \) is often equal to \( \sigma(i) - (0, 0, 1) = (i'_-, p_-, l_-) - (0, 0, 1) \). In the same way, \( \sigma(i + 1) = (i'_+, p_+, l_+) \) is often equal to \( \sigma(i) + (0, 0, 1) = (i'_+, p_+, l_+) \). The affine relations that often occur are:

\[
(i'_-, p_-, l_-) = (i'_-, p_-, l_-) - (0, 0, 1) \land (i'_+, p_+, l_+) = (i'_+, p_+, l_+) + (0, 0, 1)
\]

We separate the domain \( D' \) into two sub-domains referred to as \( D_{steady} \) and \( D_{boundary} \); the affine relations mentioned above (between \( \sigma(i) \) and \( \sigma(i - 1) \), and between \( \sigma(i) \) and \( \sigma(i + 1) \)) hold everywhere in \( D_{steady} \). For many codes of interest \( D_{steady} \) is much larger than \( D_{boundary} \). Therefore, the code generated for \( D_{steady} \) avoids expensive integer computations. We generate code for the two domains.
\[ D_{\text{steady}} = \{(i, j_0, \ldots, j_n) : i \in D, \sigma^{-1}(j_0) = u_0(i), \ldots, \sigma^{-1}(j_n) = u_n(i), j_0 = \alpha_0 j_\ell + \beta_0, \ldots, j_n = \alpha_n j_\ell + \beta_n\} \]

\[ D_{\text{boundary}} = D' - D_{\text{steady}} \]

with the same schedule as given above. The second domain allows to scan the remaining, non-steady instances that corresponds to boundary conditions. As our generated code contains explicit parallel loops going over processor dimension \( p \), we map each iteration of the \( p \) loop to a distinct physical processor to obtain parallel code. As an example, the generated 1-D jacobi code is listed as follows.

```c
void *compute(void *arg)
{
    int *start = (int *)arg;

    int t, i_prime_plus, l_plus, i_prime_minus, l_minus, p_minus, i_prime, l, p, i;

    p = *start;

    processor_bind(P_LWPID, P_MYID, *start, NULL);
    usleep(1);

    /* Generated from jacobi1_new.cloog by CLooG v0.12.2 64 bits in 0.25s. */
    for(t = 0; t <= 14; t++) {
        //Statement 1: p = 0
        if (p == 0) {
            for (l=1;l<=14;l++) {
                b[0][0][15] = a[0][0][14]+a[0][0][15]+a[1][0][0];
            }
        }
        b[0][0][15] = a[0][0][14]+a[0][0][15]+a[1][0][0];
        for (i_prime = 1; i_prime <= 3905; i_prime++) {
            b[i_prime][0][0] = a[i_prime-1][0][15]+a[i_prime][0][0]+a[i_prime][0][1];
            for (l=1;l<=14;l++) {
                b[i_prime][0][l] = a[i_prime][0][l-1]+a[i_prime][0][l]+a[i_prime][0][l+1];
            }
        }
        b[1][0][15] = a[1][0][14]+a[1][0][15]+a[1][0][0];
        b[3906][0][15] = a[3906][0][14]+a[3906][0][15]+a[0][0][0];
    }

    //Statement 1: 1 <= p <= 14
    if (p >= 1 && p <= 14) {
        b[0][p][0] = a[3906][p-1][15]+a[0][p][0]+a[0][p][1];
        for (l = 1;l <= 14; l++) {
            b[0][p][l] = a[0][p][l-1]+a[0][p][l]+a[0][p][l+1];
        }
        b[0][p][15] = a[0][p][14]+a[0][p][15]+a[1][p][0];
        for (i_prime = 1; i_prime <= 3905; i_prime++) {
            b[i_prime][p][0] = a[i_prime-1][p][15]+a[i_prime][p][0]+a[i_prime][p][1];
            for (l = 1;l <= 14;l++) {
        ```

88
\[
b[i_{\text{prime}}][p][l] = a[i_{\text{prime}}][p][l-1]+a[i_{\text{prime}}][p][l]+a[i_{\text{prime}}][p][l+1];
\]
\[
b[i_{\text{prime}}][p][15] = a[i_{\text{prime}}][p][14]+a[i_{\text{prime}}][p][15]+a[i_{\text{prime}}][p][0];
\]
\[
b[3906][p][0] = a[3905][p][15]+a[3906][p][0]+a[3906][p][1];
\]
\[
\text{for} \ (l = 1; \ l <= 14; \ l++) \ {
\text{b}[3906][p][l-1] = a[3906][p][l-1]+a[3906][p][l]+a[3906][p][l+1];
}\n\]
\[
b[3906][p][15] = a[3906][p][14]+a[3906][p][15]+a[0][p+1][0];
\]

//Statement 1: p = 15
if (p==15) {
    \text{b}[0][15][0] = a[3906][14][15]+a[0][15][0]+a[0][15][1];
    \text{for} \ (l = 1; l <= 14; l++) \ {
    \text{b}[0][15][l] = a[0][15][l-1]+a[0][15][l]+a[0][15][l+1];
    }\n    \text{b}[0][15][15] = a[0][15][14]+a[0][15][15]+a[1][15][0];
    \text{for} \ (i_{\text{prime}} = 1; \ i_{\text{prime}} < 3893; \ i_{\text{prime}}++) \ {
    \text{b}[i_{\text{prime}}][15][0] = a[i_{\text{prime}}][15][15]+a[i_{\text{prime}}][15][0]+a[i_{\text{prime}}][15][1];
    \text{for} \ (l = 1; l <= 14; l++) \ {
    \text{b}[i_{\text{prime}}][15][l] = a[i_{\text{prime}}][15][l-1]+a[i_{\text{prime}}][15][l]+a[i_{\text{prime}}][15][l+1];
    }\n    \text{b}[i_{\text{prime}}][15][15] = a[i_{\text{prime}}][15][14]+a[i_{\text{prime}}][15][15]+a[i_{\text{prime}}+1][15][0];
    }\n    \text{b}[3894][15][0] = a[3894-1][15][15]+a[3894][15][0]+a[3894][15][1];
    \text{for} \ (l = 1; l <= 16\times3894+62318; l++) \ {
    \text{b}[3894][15][l] = a[3894][15][l-1]+a[3894][15][l]+a[3894][15][l+1];
    }\n}\n
barrier_wait(&\text{barrier1});

//Statement 2: 0 <= p <= 15
for (i_{\text{prime}} = 0; i_{\text{prime}} < 3906; i_{\text{prime}}++) \ {
    \text{A}[i_{\text{prime}}][p][l] = B[i_{\text{prime}}][p][l];
}\n
barrier_wait(&\text{barrier2});

pthread_exit((\text{void} *) 0);

4.5 Evaluation

\textbf{Simulation Environment} \ We simulated a 16-core tiled CMP with the Virtutech Simics full-system simulator \cite{115} extended with timing infrastructure GEMS \cite{83}. Each tile was a 4GHz 4-way in-order SPARC processor core, with 64KB split L1 instruction and data caches, a 512KB L2 bank and a router. An on-die $4 \times 4$ mesh connected the tiles,
with 16GB one-way bandwidth per link. Cut-through routing was used in this packet-switched network. The latency on each link was modeled as 5 cycles. We had 4GB physical memory in our simulation configuration, which is larger than the memory requirement of any benchmark we used. DRAM access latency was modeled as 80ns and eight DRAM controllers were placed on the edge of the chip. The simulated hardware configuration is summarized in Table 4.1.

<table>
<thead>
<tr>
<th>Processor</th>
<th>16 4-way, 4GHz in-order SPARC cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache</td>
<td>private, 64KB I/D cache, 4-way, 64-byte line, 2-cycle access latency</td>
</tr>
<tr>
<td>L2 cache</td>
<td>shared, 8MB unified cache, 8-way, 64-byte line, 8-cycle access latency</td>
</tr>
<tr>
<td>Memory</td>
<td>4GB, 320-cycle (80ns) latency, 8 controllers</td>
</tr>
<tr>
<td>On-chip network</td>
<td>4x4 mesh, 5-cycle per-link latency, 16GB bandwidth per link</td>
</tr>
</tbody>
</table>

Table 4.1: Simulation Configuration

The coherence protocol we simulated is very close to the STATIC-BANK-DIR protocol in [84]. We adopted a similar implementation from GEMS 1.4 and made a few changes for our experiments. With this protocol, a directory is distributed across all the tiles as shown in Figure 4.1(b) such that each physical address has an implicit home tile. While our approach is not limited to this bank mapping scheme, the simulated coherence protocol interleaves cache lines.

**Benchmark Suite** While our approach can be applied in combination with program parallelization, we evaluated our approach with a set of data-parallel benchmarks as finding parallelism is not the focus of this chapter. The selected benchmarks are described in Table 5.2. Several of the benchmarks were taken from [111]. Iterative benchmarks, such as Skeleton, were set to execute 15 iterations.
Table 4.2: A data-parallel benchmark suite used in evaluation.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum</td>
<td>Compute the sum of two 1000x1000 32-bit floating-point matrices.</td>
</tr>
<tr>
<td>Mv</td>
<td>Compute matrix-vector product between a 1000x1000 matrix and a vector.</td>
</tr>
<tr>
<td>Demosaic</td>
<td>Compute an RGB image from a 1000x1000 pixel Bayer pattern, more details in [111].</td>
</tr>
<tr>
<td>Convolve</td>
<td>Convolve a 1000x1000 monochromatic image with 5x5 Gaussian filter.</td>
</tr>
<tr>
<td>Life</td>
<td>Compute Conway’s “Game of Life” on a 1000x1000 grid, more details in [111].</td>
</tr>
<tr>
<td>1D Jacobi</td>
<td>1D Jacobi computation with an iterative loop.</td>
</tr>
<tr>
<td>2D Jacobi</td>
<td>2D Jacobi computation with an iterative loop.</td>
</tr>
<tr>
<td>Skeleton</td>
<td>Compute the shape skeleton of a 2D object that is represented as non-zero pixels, used as an example in [22].</td>
</tr>
</tbody>
</table>

**Experimental Results** We implemented the localization analysis framework by taking information (index set and array access functions) from LooPo’s [79] dependence tester and generating statement-wise affine transformations. After localization analysis, the parallel code is generated by a tool based on CLooG, as presented in Section 4.4. We compiled generated C programs with GCC 4.0.4 using the “-O3” optimizing flag on a Sun SPARC Enterprise T5120 server. In order to verify the correctness of applied transformations, we wrote a validation procedure for every generated program, to compare with the result of a sequential implementation using canonical array layout. In our current implementation, we padded array elements to powers of two in bytes. For example, in Demosaic we used the RGBA format instead of the RGB format. An alternative solution to this problem is to have structures of arrays (SOA) instead of arrays of structures (AOS).

Table 4.3 summarizes the localization analysis results for the benchmarks. For each benchmarks examined, there exists at least a computation allocation and a data mapping that partially localize data accesses.

In order to assess the impact of computation allocation and data layout transformation, for each benchmark we generated two versions of parallel code:
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Communication free?</th>
<th>Fully localizable?</th>
<th>Partially localizable?</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>Mv</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>Demosaic</td>
<td>No</td>
<td>Yes</td>
<td>N/A</td>
<td>2</td>
</tr>
<tr>
<td>Convolve</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>2</td>
</tr>
<tr>
<td>Life</td>
<td>No</td>
<td>Yes</td>
<td>N/A</td>
<td>1</td>
</tr>
<tr>
<td>2D Jacobi</td>
<td>No</td>
<td>Yes</td>
<td>N/A</td>
<td>1</td>
</tr>
<tr>
<td>Skeleton</td>
<td>No</td>
<td>Yes</td>
<td>N/A</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.3: Localization analysis results of the benchmarks.

1. We divided the iteration space of each statement into $P$ chunks along the outermost parallel loop and used canonical data layout. This version corresponds to the code generated by the state-of-the-art production compilers for data-parallel languages. This version is referred to as *canon*.

2. Following the proposed localization analysis and code generation framework, we generated parallel code that uses non-canonical data layout with padding, strip-mining and permutation. This version is referred to as *noncanon*.

Figure 4.6 shows normalized remote L2 cache access numbers compared to *canon* that uses canonical data layout. As illustrated in Figure 4.6, code generated by the proposed framework significantly reduces the number of remote accesses. On average, with *noncanon* the number of remote L2 accesses is only 49% of that with *canon*. For iterative benchmarks including Life, 1D Jacobi, 2D Jacobi and Skeleton, this improvement is more significant, up to 0.039 compared with *canon*. This is because these benchmarks have most of their working sets staying in the L2 cache across iterations and benefit from greatly reduced L2 miss latencies.

Figure 4.7 shows normalized on-chip network link utilization. On average, on-chip network traffic is reduced by 81.8% from *canon*. The most significant link utilization reduction
Figure 4.6: Normalized remote L2 cache access numbers with \( \#\text{remoteAccesses}(\text{canon}) = 1 \).

Figure 4.7: Normalized link utilization of on-chip network with \( \text{utilization}(\text{canon}) = 1 \) is with Skeleton. The Skeleton code using noncanonical data layout only results in 2.39% of link utilization by the code using canonical layout. The above results have implications beyond performance. Power has become a first-order issue in chip design. Research has
shown that the on-chip interconnect contributes to up to 50% of total power consumption of a chip [82]. Several hardware/software approaches have been proposed to reduce the energy consumption from the on-chip network. [103, 30, 74] Link utilization reduction illustrated in Figure 4.7 essentially translates to reduction in dynamic power dissipation over the on-chip network.

Figure 4.8 shows speedups relative to *canon*. We can observe significant performance improvements with our approach. The average speedup in our experiments is 1.64. The best speedup achieved is 2.83 with 1D Jacobi, which also has the most significant remote L2 access reduction.

4.6 Related Work

There has been a large body of prior work addressing the problem of avoiding remote memory accesses on NUMA/DSM systems, such as [88, 23]. Although our work is related in spirit, it significantly differs from prior work in two aspects. (i) NUCA architectures
pose a very different performance issue than NUMA machines. In NUCA architectures, data with contiguous addresses are spread across L2 cache banks at a fine granularity by a static bank mapping function. Dynamic migration of cache lines between banks is not feasible due to its complexity and power consumption. In contrast, NUMA systems allocate memory at the page level by following the “first touch” rule or providing dynamic page allocation through software. This allows the programmer to control data distribution with canonical row-major or column-major layouts. (ii) To the best of our knowledge, none of the works provide a characterization as to when it is possible to achieve localizable computation allocation and data mapping.

There has been prior work attempting to use data layout optimizations to improve spatial locality in programs. Leung and Zahorjan [73] were the first to demonstrate cases where loop transformations fail and data transformations are useful. O’Boyle and Knijnenburg [89] presented techniques for generating efficient code for several layout optimizations such as linear transformations, memory layouts, alignment of arrays to page boundaries, and page replication. Kandemir et al. [60] presented a hyperplane representation of memory layouts of multi-dimensional arrays and show how to use this representation to derive very general data transformations for a single perfectly-nested loop. In the absence of dynamic data layouts, the layout of an array has an impact on the spatial locality characteristic of all the loop nests in the program which access the array. As a result, Kandemir et al. [58, 59, 60] and Leung and Zahorjan [73] presented a global approach to this problem; of these, [58] considered dynamic layouts. The motivating context and our approach to solution are fundamentally different from the above efforts.

Chatterjee et al. [26] presented a framework to determine array alignments in data-parallel languages such as HPF. While seemingly similar, the problem we address arises
from the bank mapping imposed by the architecture and our approach to code transformation for non-canonical layouts is very different. Barua et al. [9] proposed to interleave consecutive array elements in a round-robin manner across the memory banks of the RAW processor. An optimization called modulo unrolling is applied to unroll loops in order to increase memory parallelism. Extending the approach by Barua et al. [9], So et al. [105] proposed to use custom data layouts to improve memory parallelism of FPGA-based platforms.

Rivera and Tseng [101, 102] presented data padding techniques to avoid conflict misses. Within a different context, inter- and intra-array padding are employed by us to group elements into the same cache bank.

Chatterjee et al. [28, 27, 25] studied non-canonical data layouts such as 4D layout and different Morton layouts in matrix computations. The key idea is preserving locality from a 2D data space in the linear memory space. Although also employing non-canonical data layouts, our study differs from the above research in several aspects. The mismatch between data and memory space in our study is due to the banked cache organization so we may separate contiguous data in the memory space. Our focus on automatic code generation for general affine programs is very different from the above works.

Anderson et al. [8] employed non-canonical data layouts in compiling programs on shared-memory machines. Our work is related to theirs in that we also strip-mine array dimensions and then apply permutations. To avoid conflict misses and false sharing, the approach by Anderson et al. [8] maps data accessed by a processor to contiguous memory locations. In comparison, our approach attempts to spread data such that they are mapped to the same cache bank. Our data layout transformation approach shares some similarities
with their approach, but our localization analysis and code generation framework using CLooG are very different.

Lim and Lam [77, 76] previously developed a formulation for near-neighbor communication; that formulation does not apply here nor can its adaptation be reused. This is due to its being a set-and-test approach and that too only when neighbors within a fixed distance (independent of any parameters) can be found. In particular, the constraint is formulated as $\phi(i) - \phi(i') \leq \gamma$, where $\gamma$ is a constant. The approach does not address the following: (i) what is the value to be used for $\gamma$?; (ii) how is $\gamma$ varied when a solution is not found for a fixed $\gamma$; and (iii) how can the non-existence of any solution with such a constant $\gamma$ (when the difference cannot be free of program parameters) be detected. Detecting the latter case is essential for feasibility of localization itself. Our approach solves all of these problems.

4.7 Summary

Future chip multiprocessors will likely be based on a tiled architecture with a large shared L2 cache. The increasing wire delay makes data locality exploitation an important problem and it has been addressed by many hardware proposals. In this chapter, we developed a compile-time framework for data layout transformation to localize L2 cache accesses using a polyhedral model for program transformation and code generation using CLooG. Significant improvements were demonstrated on a set of data-parallel benchmarks on a simulated 16-core chip-multiprocessor.
In this chapter, we present our work on software-controlled object-level cache partitioning, which is one special form of inter-object data layout optimization.

5.1 Introduction

The performance gap between the processor and DRAM has been increasing exponentially for over two decades. With the arrival of multicore processors, this “memory wall” problem is even more severe due to limited off-chip memory bandwidth [17]. Reducing cache misses is key to achieving high performance on modern processors. In this study, we use software methods to address a weakness of LRU-based hardware management of shared last-level caches in current processors. Most cache designs are based on the LRU replacement policy and its approximations. While the LRU policy offers high performance for workloads with strong data locality, it does not identify weak-locality accesses with long reuse distances and thus often incurs cache thrashing with memory-intensive workloads. Previous studies have shown this significant problem with the LRU policy and proposed a few solutions. Despite their design differences, these hardware proposals follow one of
two directions: (1) Hybrid replacement schemes such as [108, 95] that dynamically select from multiple replacement policies based on runtime information, and (2) Cache bypassing approaches [47, 54] that identify weak-locality accesses and place them in a dedicated cache (bypass buffer) to avoid cache pollution. These approaches share one important limitation: they introduce both storage overheads and latency penalties, making it doubtful if future processors will implement them. Instead of taking transparent hardware solutions, some commercial designs have chosen to partially address the problem by providing special caching instructions such as the non-temporal store instruction on Intel architectures [52]. However such hybrid approaches are architecture-specific and limited to certain types of memory accesses such as streaming writes, and it is often not feasible for the programmer or the compiler to produce optimized code versions across all possible cache configurations and program inputs.

To address the above problem with the LRU policy, we propose a software framework that partitions the cache at the data object level to reduce cache misses for sequential programs and data-sharing parallel programs. Our approach is motivated by the following observation: many weak-locality accesses at the whole program level may have strong locality within one or a few data objects. A memory location is said to have weak locality if the reuse distance (number of distinct memory references between successive accesses to the given location) is greater than the cache capacity. By judiciously segregating objects that have interfering access patterns, we can exploit strong locality within objects in their own cache partitions using the conventional LRU policy. In this chapter we focus on partitioning the last-level cache space among large global and heap objects for high-performance scientific applications. For a given program, the proposed framework first
generates program profiles (object-relative reuse distance histograms and inter-object interference histograms) for global and heap objects using training inputs. Based on the training profiles it then detects the patterns of the program profiles. When the program is scheduled to run with an actual input, the framework predicts its locality profile based on the detected locality patterns and with actual cache configuration parameters an object-level cache partitioning decision is made to reduce cache misses. We have implemented the cache partitioning framework with a modified Linux kernel that enforces partitioning decisions by carefully laying out data objects in the physical memory. By running a set of memory-intensive benchmarks on a commodity chip multiprocessor (CMP), we show that the proposed framework is very effective in handling two important scenarios. First, when there are cache hogs, i.e. objects with unexploitable data locality, the proposed framework segregates them from the rest of the objects to improve whole-program locality. Second, when contention between multiple strong-locality objects incurs cache thrashing, the framework alleviates the problem by constraining the effective cache capacity to a subset of the objects.

The contributions of the chapter are three-fold. First, to the best of our knowledge, this chapter is the first work that uses object-level cache partitioning to reduce capacity misses for both sequential programs and OpenMP-style data-sharing parallel programs. In comparison, previous related studies [16, 19, 116] either focus on reducing conflict misses or depend on additional hardware support and modified instruction sets. Second, our approach works across program inputs and cache configurations. The proposed framework is also independent of compiler implementations by working with binary executables. Third,
our framework has been implemented and evaluated in commodity systems instead of simulation environments; therefore it can be directly used in practice to improve application performance.

The rest of the chapter is organized as follows. We first discuss a motivating example in Section 5.2. We then present an overview of the proposed object-level cache partitioning framework in Section 5.3. In Section 5.4, we introduce object-level program locality profiles used in this chapter. In Sections 5.5, 5.6 and 5.7, we describe how we generate program profiles, analyze generated profiles and make partitioning decisions based on the analysis results. We evaluate the effectiveness of our approach in Section 5.8 on a commodity CMP using programs from a computational kernel set, SPEC CPU2000 benchmarks and NAS benchmarks. We discuss related work in Section 5.9 and present our conclusions in Section 5.10.

5.2 A Motivating Example

Here we use the conjugate gradient (CG) program from the NAS benchmarks as a motivating example to illustrate the problem. As shown in Figure 5.1, CG spends most of its running time on a sparse matrix-vector multiplication \( w = a \cdot p \), where \( a \) is a sparse matrix, \( \text{rowstr} \) and \( \text{colidx} \) are row and column index arrays and \( w \) and \( p \) are dense vectors. There are also code pieces not shown in Figure 5.1 due to space limitations. These code pieces access arrays \( iv, v, acol, arow, x, r, q, z, aelt \) in addition to the arrays shown in Figure 5.1. In CG, the majority of the memory accesses are on arrays \( a \), \( p \) and \( \text{colidx} \). Although vector \( p \) has high temporal reuse in the matrix-vector multiplication code, depending on its size, its elements may get repeatedly evicted from cache before their next uses, due to the streaming accesses on arrays \( a \) and \( \text{colidx} \). As the result of this thrashing effect from accessing arrays
$a$ and $colidx$, CG often reveals a streaming access pattern and has a very high miss rate in cache. Without special code/data treatment based on domain knowledge, general compiler optimizations, such as loop tiling, cannot be applied in this case because of the irregular nature of this program — there is indirection in array accesses.

```c
for (i = 0; i < niters; i++) {
    ... ... // other code, with accesses to arrays not shown
    for (j = 1; j <= lastrow-firstrow+1; j++) {
        sum = 0.0;
        for (k = rowstr[j]; k < rowstr[j+1]; k++) {
            sum = sum + a[k]*p[colidx[k]];
        }
        w[j] = sum;
    }
    ... ... // other code, with accesses to arrays not shown
}
```

Figure 5.1: An outline of NAS-CG code.

The above problem occurs because the conventional LRU cache replacement policy does not distinguish strong- and weak-locality accesses and thus is unable to treat them differently. Since cache replacement decisions are made at the whole-program level, any data reuse with a reuse distance greater than the cache size cannot be exploited by the cache. If we allow the cache space to be partitioned between data objects, there are opportunities to better exploit temporal locality at the data object level since the reuse distances of certain accesses can be reduced without increasing the other accesses’ reuse distances. With CG, there are different ways to reduce and even completely eliminate capacity misses on array $p$ without increasing cache misses on the other objects. One approach is to protect $p$ in an exclusive cache space and leave the remaining cache capacity for the remaining data.
objects. Alternatively, we can divide the cache such that the minimum cache quota is given to arrays $colidx$ and $a$. This optimization is not limited to single-thread performance.

Even when the code is augmented with OpenMP directives, with a shared cache the above partitioning decisions should still reduce capacity misses, since memory accesses from different processor cores collectively reveal the same pattern as with sequential execution. If we allocate a very small cache quota for arrays $colidx$ and $a$ and co-schedule CG with other programs, it no longer exhibits a streaming access pattern that significantly interferes with its co-runners, so that high throughput can be achieved with judicious inter-thread cache partitioning. In this chapter, we focus on object-level cache partitioning and defer the combination of inter-object and inter-thread cache partitioning to future work.

5.3 Overview of the Approach

The CG example in Figure 5.1 demonstrates the benefits of partitioning the cache space at the object level. In this chapter the term object is defined as an allocated region of data storage and used interchangeably with variable. Note that this definition is not equivalent to its usage in object-oriented programming. We partition the last-level cache space among global and heap objects for high-performance scientific applications. There are two reasons for this decision. First, high-performance scientific applications often have relatively regular memory access patterns and high data reuse ratios, which makes object-level cache partitioning possible and profitable. Second, in these programs, the majority of the memory accesses and cache misses are on a limited number of global and heap objects. In order to partition the last-level cache space among data objects, we need to answer the following important questions: (1) How can we capture data reuse patterns at the object level, across cache configurations and program inputs? (2) How can we capture the interference among
To answer the above questions, we propose a framework that detects a program’s data reuse patterns at the object level, through memory profiling and pattern recognition, and enforces partitioning decisions at run time with operating system support. This proposed framework consists of the following steps and is summarized in Figure 5.2.

1. **Profile Generation.** For a given program and several small training inputs, we capture memory accesses in an object-relative form through binary instrumentation. We obtain object-relative reuse distance histograms and inter-object interference histograms for data objects. These histograms are program profiles with training inputs that are to be used to predict the program’s data access and reuse patterns.
2. *Profile Analysis.* Based on program profiles from training runs, we detect the patterns of the program’s object-relative data reuse, object sizes and access frequencies as polynomial functions, using a modified version of the pattern recognition algorithm by Zhong et al. [127].

3. *Cache Partitioning Decision Making and Enforcement.* When the program is scheduled to run with an actual input, we predict its object-relative reuse distance histograms and inter-object interference information with detected access patterns. We then categorize data objects as being “hot”, “hog”, “cold” or “other”. Using this classification, we follow a heuristic algorithm to make an object-level cache partitioning decision so that “hog” objects do not prevent us from exploiting the locality of “hot” objects and the contention between “hot” objects are alleviated. Such a partitioning decision is finally enforced on commodity CMPs with an OS kernel that supports *page coloring* [112] at the object level.

## 5.4 Object-Level Program Locality Profile

With a given input, we model a program’s data locality at the object level with a locality profile. An object-level program locality profile has two components: an object-relative locality profile consisting of object-relative reuse distance histograms and an inter-object interference profile including inter-object interference histograms.

### 5.4.1 Modeling Object-Relative Temporal Locality

In this chapter we focus on temporal locality at the cache line granularity because spatial locality is automatically modeled by viewing a complete cache line as the basic data unit. While this approach may appear to affect the proposed framework’s generality, it is not
a problem because our framework aims at detecting the data locality patterns of a given program binary that works across processors in the same processor family. While cache capacities and degrees of associativity often vary, processors in a modern processor family
are unlikely to use different line sizes at the same cache level. For example, Intel x86 processors with NetBurst and Core micro-architectures all use 64-byte L2 cache lines. In this chapter we exploit data locality in the last-level on-chip cache. As the last-level cache on a modern processor often has a very high degree of associativity, the impact of conflict misses is not significant and we can therefore model the cache as being fully associative.

Temporal locality is modeled using reuse distance (i.e. stack distance) [127, 18], defined as the number of distinct references between two consecutive references to the same data unit. Since in this chapter data locality is modeled with the cache line as the basic unit, reuse distance refers to the number of distinct cache lines accessed between two consecutive references to the same cache line. As it is not feasible to record the reuse distance between each data reuse, a histogram is used to summarize the temporal locality. In a reuse distance histogram, the distance space is divided into $N$ consecutive data ranges $(0, R_1], (R_1, R_2], ..., (R_{N-2}, R_{\text{max}}], (R_{\text{max}}, +\infty)$ and the value of each range represents the percentage or the absolute number of temporal reuses whose reuse distances fall into this range. $R_{\text{max}}$ is the largest cache capacity considered in terms of cache lines. While reuse distances have primarily been used for analysis at the whole program level, here we model temporal locality via a reuse distance histogram $D_A$ for accesses within each object $A$. An object-relative reuse distance histogram maintains absolute reuse counts instead of percentages, because with different inputs, objects have varying weights over the whole-program data space. As an optimization, we do not count zero distances in object-relative reuse distance histograms because such reuses are handled by the L1 cache while we optimize last-level cache accesses in this chapter.
As an example, Figure 5.3 shows simplified object-relative reuse distance histograms for objects $p$ and $a$ for the CG benchmark with input class B, on recent Intel x86 architectures with 64-byte cache lines. While the majority of $a$’s references share a large reuse distance that corresponds to $a$’s size in cache lines, 0.35% of accesses on $a$ have a very small reuse distance, in code not shown in Figure 5.1. In comparison, $p$ has relatively short reuse distances across several data ranges resulting from $p$’s random access pattern.

### 5.4.2 Modeling Inter-Object Interference

We cannot capture a program’s locality behavior with only object-relative temporal reuse information. For example, two programs can have the same object-relative temporal reuse profiles for arrays $A$ and $B$, with one accessing $A$ and $B$ in separate program sections and the other interleaving accesses on the two arrays. To compose the temporal locality information of individual objects and examine the whole-program locality behavior, we model reference interference between different objects.

Inter-object interference $I_{A,B}$ is defined as the average number of distinct data references to object $B$ per distinct reference to object $A$. Similar to temporal locality modeling, we extend the above definition to the cache line level. Note that inter-object interference is not symmetric, that is, $I_{A,B}$ and $I_{B,A}$ may not be identical. For a simple regular program, $I_{A,B}$ can be a constant. However, for complex programs with multiple phases, $I_{A,B}$ may vary with phase changes, often with changes on $A$’s reuse distances. Therefore we use a histogram to summarize inter-object interference $I_{A,B}$. In such a histogram, data ranges correspond to those in reuse distance histogram $D_A$ and the value over each range represents the average number of distinct cache lines accessed on object $B$ per distinct cache line accessed on $A$, between $A$’s reuse with a reuse distance in the range.
Figure 5.4: Examples of inter-object interference histograms for CG, generated with input class B.

For example, Figure 5.4 shows the simplified interference histograms for $I_{a,p}$ and $I_{p,a}$ for the CG benchmark with input class B. Figure 5.4(a) shows interference from object $p$ to
object $a$. $I_{a,p}$ is nearly zero although most accesses on $a$ and $p$ are interleaved. This is a consequence of the fact that between any temporal reuse of $a$, all the elements in $a$ are accessed only once while elements in the much smaller vector $p$ have much higher reuse counts. The exception is with $p$’s accesses having small reuse distances. As mentioned above, this corresponds to an infrequently executed loop. In comparison to $I_{a,p}$, Figure 5.4(b) shows that $I_{p,a}$ increases with the reuse distances of $p$. When $p$’s reuse distances are between 8000 and 10000 cache lines, the interference from $a$ to $p$ is as high as 3.91. This means that between data reuses on $p$ with reuse distance 9000 there are on average $9000 \times 3.91 = 35190$ distinct cache lines accessed on $a$. We can see from $I_{p,a}$ and $D_p$ that object $a$ significantly interferes with object $p$’s temporal reuse since a large portion of $p$’s references involves large interference values.

5.4.3 Cache Miss Estimation

As we will show in Section 5.7, the key operation in the partitioning decision algorithm is the estimation of shared cache misses on a set of objects for a given cache size. This essentially relies on merging of the object-relative reuse distance histograms and inter-object interference histograms of a set of objects. Once we have such a combined reuse distance histogram $D_S$, all the accesses with reuse distances greater than the given cache capacity in cache lines are estimated as misses. The profile combination process is as follows.

- For a reference to object $i$, its reuse distance in the combined object set $S$ is computed as:

$$dist_S = dist_i + \sum_{j \in S - \{i\}} I_{i,j}[dist_i]$$
Therefore, in an object-level reuse distance histogram, each bar is shifted to the right by a distance determined by its range and interference from the other objects in the group. The resulting reuse distance histogram for the object group is the combination of these individual right-shifted histograms.

- For an object group $S$, its combined inter-object interference with an object $k$ outside this group is computed as:

$$I_{k,S}[dist_k] = \sum_{j \in S} I_{k,j}[dist_k]$$

For example, based on the object-relative reuse distance histograms and inter-object interference histograms shown in Figures 5.3 and 5.4, we combine object-level locality information of $a$ and $p$ and obtain a combined reuse distance histogram $D_{\{a,p\}}$ in Figure 5.5. If we assign 1MB (16K cache lines) to the object group consisting of $a$ and $p$, the total number of cache misses on this object group is estimated to be $\#\text{accesses}_{\{a,p\}}(\text{reuseDist} > 16384) = 1.23 \times 10^{10}$, where $\#\text{accesses}_{\text{obj}}(\text{reuseDist} > R)$ denotes the number of references to object $\text{obj}$ with reuse distances larger than $R$.

Figure 5.5: The combined reuse distance histogram for object group $\{a, p\}$ in CG with input class B.
5.5 Profile Generation

Figure 5.6 illustrates how a program profile with a training input is generated. There are three important components used in profile generation: object table, custom memory allocator and memory profiler.

Object Table The object table maintains the basic information of every profiled object. As the hub of the profiling process, it is updated and queried by both the custom memory allocator and the memory profiler. Object information stored in the object table includes object identifier, name, starting address, size, location and liveness. In this chapter we focus on global and heap objects therefore an object’s location is either heap or global. An object’s identifier is used to facilitate fast query and retrieval. A global object’s identifier
is decided by its order in the symbol table of the binary executable. A heap object’s identifier is calculated by a hash function that takes its allocation site, allocation order and the total number of global objects as parameters. Identifier 0 is reserved for the special object \( obj_0 \) that includes all the data not explicitly profiled. Similar to identifiers, object names are retrieved from the symbol table for global objects and decided by a function mangling allocation sites and allocation order for heap objects. Because heap objects may have overlapping address ranges due to their non-overlapping life cycles, a raw address can be found within multiple heap objects. We keep track of each object’s liveness to avoid this problem. In this way during profile generation each memory access only affects a live object’s object-relative temporal locality information and inter-object interference information.

**Custom Memory Allocator**  The custom memory allocator is used to capture each heap object’s creation and deletion. We replace standard memory management functions such as malloc(), calloc(), free(), and realloc() with our implementations. In this way during profiling runs memory management requests are redirected to the memory profiler and heap objects’ life cycles are tracked in the object table.

**Memory Profiler**  The memory profiler controls the profiling process and starts a training run by updating the object table with global object information from the executable’s symbol table. It relies on binary instrumentation to obtain the raw address stream of a given program. Our current profiler implementation is written as a tool based on PIN [80] that inserts instruction and object probes before every instruction accessing the memory. The core components in the profiler are a set of *reuse distance profilers* and an *inter-object interference counter table*. A reuse distance profiler is used to track the reuse distances of an object and implemented with a hash table and a Splay tree following Sugumar and
Abraham [109]. The inter-object interference counter table is two-dimensional and used to track inter-object interference between temporal reuses. As in high-performance scientific programs the majority of the accesses are on large global and heap objects and it is not feasible to include all objects in cache partitioning decision making, we only track track objects larger than a threshold (2KB) in memory profiling. Small objects are merged into a special object \( obj_0 \). The memory profiling algorithm used is summarized in Algorithm 5.1.

---

**Algorithm 5.1** The memory profiling algorithm.

Require: \( objProfiler.trace(addr) \) returns the reuse distance of access \( addr \) on object \( obj \) and \( hist.sample(d, n) \) collects \( n \) data with value \( d \) into histogram \( hist \).

```
1: \( \text{tracedAddr}[0..\text{objNum} - 1] \leftarrow 0 \)
2: \textbf{for} each memory reference with raw address \( addr \) \textbf{do}
3: \hspace{1em} Search the object table for such a live object \( obj \) that \( obj.startAddr \leq addr \) and \( obj.startAddr + obj.size \geq addr \)
4: \hspace{1em} \textbf{if} \( obj \) exists \textbf{then}
5: \hspace{2em} \( (\text{objID}, \text{offset}) \leftarrow (\text{obj.ID}, (addr - \text{obj.startAddr})/\text{CacheLineSize}) \)
6: \hspace{1em} \textbf{else}
7: \hspace{2em} \( (\text{objID}, \text{offset}) \leftarrow (0, \text{addr}/\text{CacheLineSize}) \)
8: \hspace{1em} \( \text{reuseDist} \leftarrow \text{reuseDistProfiler}[\text{objID}].\text{trace}(\text{offset}) \)
9: \hspace{1em} \( \text{objReuseDistHistogram}[\text{objID}].\text{sample}(\text{reuseDist}, 1) \)
10: \hspace{1em} \textbf{if} \( \text{tracedAddr}[\text{objID}] = 0 \) \textbf{then}
11: \hspace{2em} \( \text{tracedAddr}[\text{objID}] \leftarrow \text{addr} \)
12: \hspace{1em} \textbf{continue} to process next memory access
13: \hspace{1em} \textbf{if} \( \text{tracedAddr}[\text{objID}] = \text{addr} \) \textbf{and} \( \text{reuseDist} \neq 0 \) \textbf{then}
14: \hspace{2em} \( \text{tracedAddr}[\text{objID}] \leftarrow 0 \)
15: \hspace{1em} \textbf{for} each \( i \) where \( i \neq \text{objID} \) \textbf{do}
16: \hspace{2em} \textbf{if} \( \text{interferenceCounter}[\text{objID}][i] \neq 0 \) \textbf{then}
17: \hspace{3em} \( \text{interferenceHistogram}[\text{obj}][i].\text{sample}(\text{reuseDist}, \)
18: \hspace{3em} \( \text{interferenceCounter}[\text{objID}][i]/\text{reuseDist} \)
19: \hspace{3em} \( \text{sampleCounterHistogram}[\text{obj}][i].\text{sample}(\text{reuseDist}, 1) \)
20: \hspace{2em} \( \text{interferenceCounter}[\text{objID}][i] \leftarrow 0 \)
21: \hspace{1em} \textbf{for} each \( j \) where \( j \neq \text{objID} \) \textbf{do}
22: \hspace{2em} \textbf{if} \( \text{reuseDist} > \text{interferenceCounter}[\text{objID}][j] \) \textbf{then}
23: \hspace{3em} \( \text{interferenceCounter}[j][\text{objID}] \leftarrow \text{interferenceCounter}[j][\text{objID}] + 1 \)
24: \hspace{1em} \textbf{for} each \( i \) where \( i \neq j \) \textbf{do}
25: \hspace{2em} \textbf{for} each range \( k \) \textbf{do}
26: \hspace{3em} \( \text{interferenceHistogram}[i][j][k] \leftarrow \frac{\text{interferenceCountHistogram}[i][j][k]}{\text{sampleCounterHistogram}[i][j][k].\text{range}[k]} \)
```

---

114
As the cost of binary instrumentation is high, for each profiled object, we optimize the memory profiler by collecting reuse distances for a portion of the accesses and then estimate the complete reuse distance histogram based on the sampled profile. Note that every access is still recorded for later reuse distance collection and inter-object interference counting. We have tried different sampling ratios and found in practice 10% as the best trade-off between accuracy and profiling cost. Compared with complete profiling, the error introduced by sampling is less than 2% in our experiments. By sampling 10% of reuse distances and employing several other optimizations, our current implementation of the memory profiler has a slowdown of 50 to 80 times. This profiling cost, while still seemingly high, can be easily amortized over the lifetime of an executable, as the program complexity is often quadratic or cubic and profiling runs are made with much smaller problem sizes than actual runs that is of interest to optimize.

5.6 Profile Analysis

After at least two program profiles with different training inputs are obtained from the profiling process, object-relative data locality patterns are detected following an algorithm similar to the approach by Zhong et al. [127]. The idea is as follows. Each object-relative reuse distance histogram is divided into \( n \) small groups. With two histograms \( p_1 \) and \( p_2 \) corresponding to different input sizes, we need to find a pattern function \( f_k \) to fit each formed group in two profiles, \( g_{1,k} \) and \( g_{2,k} \), for \( k \) from 1 to \( n \). \( f_k \) matches average reuse distances \( d_{1,k}, d_{2,k} \) in group \( k \) such that \( f_k(x_1) = d_{1,k} \) and \( f_k(x_2) = d_{2,k} \), where \( x \) is a program parameter. In this chapter, for \( f_k \) we consider simple polynomial functions in the form of \( f_k(x) = a_k + b_k x^m \), where \( m = 0, 1, 2, 3, 4 \) and \( a_k, b_k \) are constants. In order to detect the locality pattern function \( f_k \) for each group \( k \), we first pick an exponent \( m \) such
that $x_1^m/x_2^m$ is the closest to $d_{1,k}/d_{2,k}$. Then we decide $a_k$ and $b_k$ by solving equations $d_{1,k} = a_k + b_k x_1^m$ and $d_{2,k} = a_k + b_k x_2^m$. Following the above process an object’s locality pattern is summarized as a list of polynomial functions $f = \{f_1, f_2, \ldots, f_n\}$. Note that our choice of using polynomial functions in locality pattern recognition differs from the approach by Zhong et al. that takes functions such as square root. This is mainly because we use one program parameter instead of the object size as the variable. We also follow procedures similar to locality pattern recognition to detect the patterns of the data access volume and the data size of each object as well, as absolute counts instead of percentages are kept in histograms. With all the above program patterns detected, given a new input we can construct its object-relative reuse distance histograms without tracing the actual execution. For inter-object interference histograms, we detect their patterns similar to object-level reuse pattern recognition. As an example, after pattern recognition using two training profiles, we find that in CG the majority of array $p$’s reuse distances grow linearly with program parameter $na$ while the rest of the reuse distances keep a constant pattern. In contrast, although $colidx$ and $a$’s reuse distances actually grow with the input size, they are simply predicted as large constants as they exceed the maximum cache capacity considered (8MB) even with training inputs. We also predict that inter-object interference $I_{a,p}$ is constant 0 within most ranges while $I_{p,a}$ mostly follows a linear pattern.

5.7 Cache Partitioning Decision Making and Enforcement

When the program is scheduled to run with an actual input, we first predict the program profile including object-relative reuse distance histograms and inter-object interference histograms with patterns recognized during profile analysis. The partitioning decision maker then selects objects to be isolated from the rest of the data space and decide their cache...
quotas. Finally this decision is enforced by OS via virtual-physical address mapping. Figure 5.7 illustrates the cache partitioning decision making and enforcement process. This phase of the framework shares several components with the profile generation process such as the object table and the custom memory allocator but with a few changes. For example, in the object table field \textit{colors} is added to denote the cache space allocated to an object as the total cache capacity is divided into a set of \textit{colors}(regions). Unlike the custom memory allocator used in profile generation, the memory allocator used here reads the object table to check and enforce cache partitioning decisions but never updates the table. The rest of this section discusses in detail the cache partitioning mechanism and the partitioning decision making algorithm.
5.7.1 Partitioning Enforcement

The most straightforward way to enforce cache partitioning decisions is through hardware support. Cache can be partitioned at different granularities such as cache lines, cache ways or pages. Because hardware-based partitioning support is not readily available, here we provide a software mechanism that essentially emulates page-level hardware cache partitioning based on a well accepted OS technique called page coloring [112]. It works as follows. A physical address contains several common bits between the cache index and the physical page number. These bits are referred to as page color. A physically addressed cache is therefore divided into non-overlapping regions by page color, and pages in the same color are mapped to the same cache region. We assign different page colors to different objects, thus the cache space is partitioned among objects. By limiting the physical memory pages of an object within a subset of colors, the OS can limit the cache used by the object to the corresponding cache regions. In our experiments, the Intel Xeon processor used has 4MB, 16-way set associative L2 caches, each shared by two cores. As the page size is set to 4KB, we can break the L2 cache to 64 colors (cache size / (page size × cache associativity)).

Our implementation is based on the Linux kernel. We maintain a page color table for threads sharing the same virtual memory space to guide the mapping between virtual and physical pages. Each entry in the table specifies the page color that a virtual page can be mapped to. The number of table entries is decided by the maximum number of virtual pages, which is $2^{20}$ in the 32-bit systems used in the experiments. Each thread has a pointer in its task structure pointing to the page color table. We also modify the buddy system in the memory management module of the Linux kernel, which is in charge of mapping virtual pages to physical pages, to follow the guidance specified in the page color table. We add a
set of system calls to update the page color table at user level. These system calls are used by the partitioning decision maker and the memory allocator to enforce cache partitioning decisions for global and heap objects. We reserve a special value ($0xFF$) for page color table entries to indicate the use of the default random page mapping policy. As we initialize the page color table using this special value, programs not involved in object-level cache partitioning simply keep their default behaviors on a unmodified Linux kernel.

Because all the threads in a thread family share the same virtual memory space, they share the same page color table. In our implementation, each table entry occupies one byte to represent the color assigned to a virtual page and therefore the page color table incurs no more than 1MB space overhead for each process family in 32-bit systems. Therefore our implementation has a negligible space overhead. Because all the decisions are made statically at user level, there is no run-time overhead in the kernel.

5.7.2 Partitioning Decision Making

Preprocessing: Object Categorization and Object Clique Search  With the predicted program profile and the actual cache configuration, we categorize objects into four types based on their temporal locality profiles:

1. **Cold objects** refer to the objects that have few memory accesses while still traced in the profiling process because of their large sizes. We set a threshold $T_{\text{cold}}$ to detect cold objects. For an object $obj$, if $\frac{\#\text{accesses}_{obj}}{\#\text{totalAccesses}} < T_{\text{cold}}$, it is categorized as a cold object. $\#\text{accesses}_{obj}$ is the number of accesses to the object $obj$ and $\#\text{totalAccesses}$ is the total number of accesses to all the objects. In our experiments, we set $T_{\text{cold}}$ as 1%. 

119
2. **Hog objects** refer to the objects that have high memory demands but reveal little or no temporal locality. With a threshold $T_{hog}$, if an object $obj$ is not a cold object and

$$\frac{\#\text{accesses}_{\text{obj}}(\text{reuseDist} \leq \text{cacheSize})}{\#\text{accesses}_{\text{obj}}} < T_{hog},$$

then it is a hog object. Note that $\text{cacheSize}$ is in terms of cache lines. $\#\text{accesses}_{\text{obj}}(\text{reuseDist} \leq R)$ refers to the number of references to object $obj$ with reuse distances less than or equal to $R$ and we have $T_{hog} = 2\%$.

3. **Hot objects** are the objects with high temporal locality. For an object $obj$, if we have

$$\frac{\#\text{accesses}_{\text{obj}}(\text{reuseDist} \leq \text{cacheSize})}{\#\text{accesses}_{\text{obj}}} > T_{hot}$$

and if $obj$ is not a cold object, then it is a hot object. We have $T_{hot} = 10\%$.

4. **Other objects** are the objects that do not belong in any above category.

For example, after object categorization, profiled objects in CG are categorized into groups, as shown in Table 5.1. Because the majority of the objects are categorized as cold objects, it significantly simplifies the partitioning decision making process.

<table>
<thead>
<tr>
<th>Cold Objects</th>
<th>Hog Objects</th>
<th>Hot Objects</th>
<th>Other Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>$iv,v,acol,arow,rowstr,x,w,r,q,z,aelt$</td>
<td>$a, colidx$</td>
<td>$p$</td>
<td>$obj_0$</td>
</tr>
</tbody>
</table>

Table 5.1: Categorizing objects in CG’s profiles.

A set of objects and their interference relationships can be viewed as a graph with vertices representing objects and unweighted edges representing interferences between objects. Based on the graph representation, we enumerate the cliques inside hog objects. As we will show in the partitioning decision algorithm, the cumulative object size of the maximum clique decides the memory requirement of all hog objects and thus decides their
cache allocation. While the clique enumeration problem has exponential space and time complexities, it does not bring much overhead in our particular problem because the number of hog object cliques is always fairly small in real programs. For example, CG has only one hog object clique that includes \( a \) and \( colidx \).

**The Partitioning Decision Making Algorithm** The partitioning decision making algorithm finds a cache partition among objects with predicted object-level reuse distances, inter-object interferences and cache configuration information. With a data-sharing parallel program, the algorithm uses its sequential counterpart to approximate its data access patterns. The partitioning decision making algorithm consists of four major steps. (1) To simplify late parts of the algorithm, we first merge the reuse distance histograms and inter-object interference histograms of \( cold \) and \( other \) objects with those of the special object \( obj_0 \). (2) Although in theory hog objects do not need any cache space, we still need to allocate enough cache capacity to them because a physically addressed cache and the physical memory are co-partitioned by the page color-based partitioning enforcement mechanism. (3) Finding the optimal cache partition is NP-hard because the decision problem of integer linear programming can be reduced to this problem. Since it is not feasible to search for the best partitioning decision in a brute-force way, a heuristic is employed to maximize the benefit-cost ratio at every cache allocation step until there is no further benefit of cache partitioning. (4) As an important optimization, once there is no additional benefit from segregating complete objects, we attempt to divide a remaining object into two segments and keep one segment in cache. In this way we exploit temporal locality even if object-level reuse distances are greater than cache size, a case where both the traditional LRU policy and object-level cache partitioning restricted to complete objects cannot handle. We only apply
this optimization to objects whose reuse distances share the same pattern with their object sizes as we notice this pattern is very common in memory-intensive scientific programs. More complex cases can be handled by further extending this approach. For example, an object of $n^3$ elements can be divided into $2n$ segments if its reuse distance $n^2$ is greater than the cache size. However, such complicated extensions have not been implemented. The complete partitioning decision making algorithm is summarized in Algorithm 5.2.

Inaccuracy exists in the data locality model as it is impractical to include architectural complexities such as prefetching and out-of-order execution. Therefore we may mistakenly choose a cache partition that does reduce cache misses in practice. As a practical solution, at each iteration of steps 3 and 4 we accept the partition only if the predicted last-level miss reduction is above a threshold compared to the previous iteration. Otherwise we stop further applying cache partitioning. In practice, we set this threshold to be 5%.

For example, when we apply Algorithm 5.2 to CG with a given input on a machine with a 4MB L2 cache, the decision making algorithm always first segregates hog objects $a$ and $colidx$ with the rest of the objects. Then our algorithm stops trying to further partition the remaining cache space as it cannot significantly reduce the number of cache misses with the remaining objects.

5.8 Experimental Results

We used the object-level cache partitioning framework to improve the L2 cache performance of a commodity system. In this section we present the experimental results.

**Experimental Environment** We conducted experiments on a Dell PowerEdge 1900 workstation with two quad-core 2.66GHz Xeon X5355 Processors and 16GB physical memory with eight 2GB dual-ranked Fully Buffered DIMMs (FB-DIMM). Each X5355 processor
**Algorithm 5.2** The partitioning decision making algorithm.

**Require:** misses(c, S) returns the cache miss number of object group S with cache allocation c, using predicted object-relative reuse distance histograms and inter-object interference histograms.

1: **STEP 1.** (Merge cold and other objects)
2: All cold and other objects are merged into obj₀
3: **STEP 2.** (Find the minimum cache space for hog objects)
4: Find the clique in hog objects with the largest memory requirement \( \text{mem}_{\text{hogs}} \)
5: \( \text{hogCacheColors} \leftarrow \lceil \frac{\text{mem}_{\text{hogs}}}{(\text{totalMemory}/\#\text{pageColors})} \rceil \)
6: \( \text{cacheColors} \leftarrow \text{totalCacheColors} - \text{hogCacheColors} \)
7: **STEP 3.** (Heuristic-based cache partitioning for hot objects)
8: \( \text{partitionedObjs} \leftarrow \phi, \text{objsLeft} \leftarrow \text{hotObjs} \)
9: **while** \( \text{objsLeft} \neq \phi \) **do**
10: \( \text{bestBenefitCost} \leftarrow 0 \)
11: **for** each object \( \text{obj} \) in \( \text{objsLeft} \) **do**
12: **for** colors \( = \lceil \frac{\text{mem}_{\text{obj}}}{(\text{totalMemory}/\#\text{pageColors})} \rceil \) to \( \text{cacheColors} \) **do**
13: Try to find non-conflicting colors in assigned colors to \( \text{partitionedObjs} \). The objects already assigned these colors should have no interference with \( \text{obj} \).
14: \( \text{cost} \leftarrow (\text{colors} - \text{nonConflictingColors}) \)
15: \( \text{benefit} \leftarrow \text{misses}(\text{cacheColors}, \text{objsLeft}) - (\text{misses}(\text{cacheColors} - \text{cost}, \text{objsLeft} - \text{obj}) + \text{misses}(\text{colors}, \text{obj})) \)
16: **if** \( \text{benefit}/\text{cost} > \text{bestBenefitCost} \) **then**
17: \( \text{bestBenefitCost} \leftarrow \text{benefit}/\text{cost} \)
18: \( (\text{obj}_{\text{best}}, \text{colors}_{\text{best}}) \leftarrow (\text{obj}, \text{colors}) \)
19: **if** \( (\text{obj}_{\text{best}}, \text{colors}_{\text{best}}) \) is not empty **then**
20: \( \text{partitionedObjs} \leftarrow \text{partitionedObjs} \cup \{\text{obj}_{\text{best}}\} \)
21: \( \text{objsLeft} \leftarrow \text{objsLeft} - \{\text{obj}_{\text{best}}\} \)
22: \( \text{cacheColors} \leftarrow \text{cacheColors} - \text{colors}_{\text{best}} \)
23: **else**
24: Break from the while loop
25: **STEP 4.** (Partial-object cache partitioning)
26: \( \text{objsLeft} \leftarrow \) objects in \( \text{objsLeft} \) with object-size reuse distances
27: Repeat lines 9-29, at each iteration candidate object \( \text{obj} \) is tentatively divided into two objects \( \text{obj}_1 \) and \( \text{obj}_2 \) where \( \text{obj}_1.\text{size} = (\text{colors} \times \text{cacheSize})/\#\text{pageColors}, \text{obj}_2.\text{size} = \text{obj}.\text{size} - \text{obj}_1.\text{size} \)

123
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input parameter</th>
<th>Input range</th>
<th>Num. of inputs</th>
<th>Objects profiled</th>
<th>Object-level reuse distance patterns</th>
<th>Num. of threads</th>
<th>Max. speedup</th>
<th>Avg. speedup</th>
<th>Max. miss reduction</th>
<th>Avg. miss reduction</th>
<th>Num. of inputs with improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>jacobi2d</td>
<td>N</td>
<td>[200,1200]</td>
<td>11</td>
<td>3</td>
<td>0,1,2,3</td>
<td>1</td>
<td>1.27</td>
<td>1.11</td>
<td>46.3%</td>
<td>20.6%</td>
<td>7</td>
</tr>
<tr>
<td>stencil3d</td>
<td>N</td>
<td>[50,100]</td>
<td>11</td>
<td>3</td>
<td>0,1,2,3</td>
<td>1</td>
<td>1.36</td>
<td>1.10</td>
<td>39.3%</td>
<td>16.0%</td>
<td>7</td>
</tr>
<tr>
<td>adv</td>
<td>NMAX</td>
<td>[200,1200]</td>
<td>11</td>
<td>4</td>
<td>0,1,2</td>
<td>1</td>
<td>1.03</td>
<td>1.01</td>
<td>24.7%</td>
<td>7.9%</td>
<td>8</td>
</tr>
<tr>
<td>fdtd2d</td>
<td>N=N1=N2</td>
<td>[200,1200]</td>
<td>11</td>
<td>4</td>
<td>0,1,2</td>
<td>1</td>
<td>1.91</td>
<td>1.20</td>
<td>78.0%</td>
<td>24.5%</td>
<td>8</td>
</tr>
<tr>
<td>swim</td>
<td>N=N1=N2</td>
<td>[128,1334]</td>
<td>9</td>
<td>15</td>
<td>0,1,2</td>
<td>1</td>
<td>1.04</td>
<td>1.01</td>
<td>3.1%</td>
<td>1.0%</td>
<td>3</td>
</tr>
<tr>
<td>art</td>
<td>objects</td>
<td>[10,50]</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>1.29</td>
<td>1.11</td>
<td>36.9%</td>
<td>16.6%</td>
<td>7</td>
</tr>
<tr>
<td>CG</td>
<td>na</td>
<td>[75000,210000]</td>
<td>10</td>
<td>15</td>
<td>0</td>
<td>1</td>
<td>1.09</td>
<td>1.07</td>
<td>35.7%</td>
<td>27.9%</td>
<td>10</td>
</tr>
<tr>
<td>LU</td>
<td>PROBLEM_SIZE</td>
<td>[60,102]</td>
<td>9</td>
<td>9</td>
<td>0,1,2,3</td>
<td>1</td>
<td>1.28</td>
<td>1.09</td>
<td>43.4%</td>
<td>14.9%</td>
<td>7</td>
</tr>
<tr>
<td>apsi</td>
<td>N=NX=NY=NZ</td>
<td>[64,112]</td>
<td>6</td>
<td>2</td>
<td>0,1,2,3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mgrid</td>
<td>$2^{LM}$</td>
<td>[32,128]</td>
<td>3</td>
<td>4</td>
<td>0,1,2,3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BT</td>
<td>PROBLEM_SIZE</td>
<td>[60,102]</td>
<td>9</td>
<td>15</td>
<td>0,1,2,3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FT</td>
<td>N=NX=NY=NZ</td>
<td>[64,512]</td>
<td>4</td>
<td>9</td>
<td>0,1,2,3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SP</td>
<td>PROBLEM_SIZE</td>
<td>[60,102]</td>
<td>9</td>
<td>15</td>
<td>0,1,2,3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.2: Characteristics and experimental results for selected benchmarks. For brevity 0,1,2,3 are used to represent constant, linear, square and cubic functions respectively.
has two pairs of cores and cores in each pair share a 4MB, 16-way set associative L2 cache. Each core has a private 32KB L1 instruction cache and a private 32KB L1 data cache. Both adjacent line prefetch and stride prefetch are enabled on this machine. Because we target shared cache performance, in our experiments we used at most a pair of cores via function `sched_setaffinity` that sets process/core affinity. Our cache partitioning mechanism was implemented in Linux kernel 2.6.20.3. While there are 64 page colors in the shared L2 cache, we only used 5 least significant color bits in a physical address. Therefore we have 32 colors and each color corresponds to 128KB of cache space. Without incurring page swapping, the maximum physical memory mapped to a page color is 512MB on this machine. We used `pfmon` [49] to collect performance statistics such as L2 cache misses.

**Benchmark Selection** We selected a set of scientific programs from a suite of computational kernels (the examples in [91]), an OpenMP implementation of NAS benchmarks [1], and the SPEC CPU2000 benchmark set [107]. These benchmarks include `jacobi2d`, `stencil3d`, `adi` and `fdtd2d` from the kernel suite, four benchmarks `apsi`, `swim`, `art` and `mgrid` from SPEC CPU2000 that were also used in a related study [106] and five benchmarks `CG`, `LU`, `BT`, `FT` and `SP` from NAS benchmarks version 2.3. We excluded `twolf` and `vpr` that were also used in [106] and `EP` and `IS` in NAS benchmarks, because they are not scientific programs and optimizing these programs is out of the scope of this study. We also did not include `MG` from the NAS benchmarks because it is essentially identical to `mgrid` from SPEC CPU2000. The selected benchmarks were compiled with Intel C/Fortran Compiler 10.1 using the “-fast” and “-openmp” flags.

The efficiency of object-level cache partitioning essentially relies on program information that distinguishes different data objects. In some cases, to fully test our framework, we
had to make source code changes in the selected benchmarks due to some limiting factors. First, some Fortran programs use common blocks which makes global objects in a common block indistinguishable. For this reason we modified array declarations in *swim* such that every global object was only in one common block. We also chose to use a C implementation of the NAS benchmarks instead of the original Fortran programs to avoid this complication. Second, some C programs use a programming idiom that creates a multi-dimensional heap array from many dynamically allocated sub-arrays. We can modify such code by allocating memory to the array at once. Such a change is needed for *art* from Spec CPU2000. Note that there is no difference in performance and memory requirement before and after such simple changes. Third, some legacy programs use a large fixed-size array as the work array for memory management. *apsi* from SPEC CPU2000 is such an example. With these programs, it is not feasible to distinguish accesses on different objects. Therefore as shown in Table 5.2, we were not able to optimize *apsi*.

The resulting benchmark suite consists of thirteen programs, including five NAS benchmarks that run both sequentially and in parallel. Their characteristics are summarized in Table 5.2. Table 5.2 also shows the range of the inputs used for each benchmark. If there is a reference input size for a benchmark, it was always used in our experiments. The two smallest inputs for each benchmark were used in both training and actual runs, while the remaining input sets were only used for the actual runs. While the profiling time is non-trivial, after two program profiles are generated and analyzed, the time taken for the cache partitioning decisions is negligible – of the order of tens of milliseconds. In the experiments, we did not observe any increase in OS activities from the kernel modification. Table 5.2 also shows the number of objects profiled for each benchmark. It can be seen that the number of profiled objects in these scientific applications is quite small – up to 15.
in the experiments. This indicates that it is feasible to effectively reason about data locality at the object level. Furthermore, the majority of the data accesses exposed by these objects are very regular and thus can be modeled using simple polynomial functions.

Table 5.2 shows performance and cache miss reduction data for our approach compared to that with a standard shared LRU-based L2 cache. For a given input, we ran each benchmark three times with the shared LRU-based L2 cache and three times with object-level cache partitioning. We then reported the speedup and cache miss reduction with the worst performance numbers with object-level cache partitioning and the best numbers with the shared LRU-based L2 cache. If the partitioning decision algorithm did not choose to partition the cache between objects, we simply set speedup to 1 and miss reduction to 0 as the shared LRU-based L2 cache was nevertheless used and the performance differences were due to measurement noise. In our experiments, we achieved performance improvements on eight benchmarks, with up to 1.91 speedup and up to 78% cache miss reduction (with \( f_{dtd2d} \) when \( N = 500 \)). In experiments involving these eight benchmarks, we improved 74 out of 100 total cases. We were unable to improve the performance of the five remaining benchmarks. In all the cases where we could not improve performance, the algorithm did not make any bad cache partitioning decision that caused performance degradation.

The inability of the proposed framework to achieve performance improvement for a few programs is due to two reasons: (1) Indistinguishable data objects in a program can make the framework lose optimization opportunities. As discussed above, apsi is such an example. (2) An object-level cache partitioning decision is a function of the program, the program input, and the cache capacity. Our approach can be effective for a program with a given input only when there is an object with reuse distances larger than the cache capacity with the conventional LRU-based cache and less than the capacity of the assigned
cache portion within an object-level cache partition. In most cases where our approach is unable to improve cache performance, reuse distances are either smaller than the cache size at the whole program level (in which case standard LRU cache is very effective) or much larger than the cache size even within one object (in which case object-level partitioning still cannot help). For example, with reference input $2^{LM} = 128$, we could not improve mgrid’s performance but the same program’s L2 miss rate was reduced in [106]. The reason is that the optimization opportunity for this program is in the accesses with $x^2$ locality patterns, whose reuse distances are slightly larger than 512KB in cache lines with conventional LRU caching but less than 512KB in cache lines within individual objects. As our experimental platform has a 4MB L2 cache while the L2 cache capacity on the PowerPC 970FX processor used in [106] was only 512KB, these accesses do not incur cache misses on our experimental platform with conventional LRU caching but could be optimized for the PowerPC 970FX in their experiment.

**Case Studies**  We analyze the effect of object-level cache partitioning in detail using CG and LU. These parallel benchmarks are interesting as they are quite complicated and both incur high miss rates across input ranges and thread numbers. Furthermore, they represent two types of applications that can benefit from object-level cache partitioning.

Figure 5.8 shows speedup and cache miss reductions with CG in comparison to the shared LRU cache. For CG, with different thread numbers and input sizes, the partitioning decision is unchanged. It always segregates hog objects colidx and a, and lets the other objects share the remaining cache capacity. Performance improvement and miss rate reduction are also across thread numbers and input sizes, as shown in Figure 5.8. When the number of threads is increased from one to two, the relative benefit of cache partitioning
Figure 5.8: CG with object-level cache partitioning in comparison to uncontrolled LRU caching.

decreases. For example, when two threads are used, CG’s average speedup is reduced from 1.07 to 1.04. Experimental data show that CG’s total L2 misses increase with the number of threads. Since the cache miss rates for $a$ and $colidx$ do not change, this reduction in
Figure 5.9: LU with object-level cache partitioning in comparison to uncontrolled LRU caching.

Performance improvement is likely due to increasing intra-object cache contention on $p$ between two threads.
Figure 5.9 shows speedups and cache miss reductions on $LU$, a program of over 3000 lines. $LU$ achieves very high performance improvements with certain input ranges using our proposed framework. For instance, when the input size is 70, it achieves a performance improvement of 31.7%, compared to uncontrolled LRU caching. However, unlike $CG$, $LU$ does not exhibit this trend of performance improvement across all input sizes. When the problem size is greater than 90, cache partitioning is not beneficial. This difference between $CG$ and $LU$ results from the fact that the majority of $LU$’s data references are on hot objects $u$, $rsd$, $a$, $b$, $c$ and $d$, not on hog objects as in $CG$. Given a cache configuration, the thrashing effect from accessing multiple hot objects is significant only over a range of input sizes. When the input goes beyond a certain value, accesses on a single hot object start to thrash in the cache. As a result, $LU$’s partitioning decisions vary with inputs and cache sizes by involving different numbers of hot objects.

**Effect of Partial Object Cache Partitioning** The partitioning decision algorithm considers dividing objects with object-size reuse distances. In the eight benchmarks with performance improvement, six of them have partial object cache partitioning with at least one input. This is because most of the benchmarks used are iterative scientific programs and large objects often have reuse distances linearly proportional to its size. For example, Figure 5.10 shows the speedup and cache miss reductions on $art$ with and without partial object cache partitioning. With $art$, the majority of its memory accesses involve three hot objects $f1.layer$, $tds$ and $bus$. When input parameter $objects$ is larger than 40, by only segregating complete objects the proposed framework cannot improve performance. However with partial object cache partitioning, object $bus$ is divided into two segments and the object’s temporal locality is exploited for the segment staying in cache.
5.9 Related Work

Previous studies have shown that the conventional LRU policy is ineffective in exploiting reuses with large distances and have proposed a few hardware solutions. The hardware proposals follow one of two directions: (1) Hybrid replacement schemes such as [108, 95] that dynamically select from multiple replacement policies based on runtime information, and (2) Cache bypassing approaches [47, 54] that identify weak-locality accesses and place
them in a dedicated cache (bypass buffer) to avoid cache pollution. In comparison, we propose a software framework that partitions the cache at the object level to reduce cache misses, requiring no architectural support.

Some commercial designs have chosen to partially address the problem with weak-locality accesses by providing special caching instructions to handle streaming accesses. However such hybrid approaches are architecture-specific and limited to certain types of memory accesses. For example, there is a non-temporal store instruction on Intel architectures [52]. However a non-temporal load instruction is not provided for cacheable memory regions on recent Intel architectures as it significantly complicates the coherence protocol [2]. Also it is often not feasible for the programmer or the compiler to produce code versions across all possible cache configurations and program inputs. In comparison, our approach works across programs and inputs without special caching instructions. In particular, our approach can resolve the cache contention between strong-locality objects, as shown by the results on NAS-LU in the chapter. We are not aware of any previous hardware/hybrid approach with such capability.

Many approaches have been proposed to partition the shared cache at the thread or process level. Most of them add cache partitioning support in the micro-architecture [78, 110, 96]. Several studies have highlighted the issues of QoS and fairness [65, 50, 24]. There have been studies on OS-based cache partitioning policies and their interaction with micro-architecture support [98, 32]. Our work differs from these studies in that we focus on object-level cache partitioning and do not require any new architectural support. There has been prior work [43] using job pairing to reduce the effect of inter-thread contention. In comparison, we focus on improving the performance of sequential and parallel collaborative workloads that may not have co-runners. Because our proposed object-level
cache partitioning framework works orthogonally to the above thread-level techniques, our framework is complementary to these techniques when there are a large number of cores sharing the cache.

Understanding data locality is critical to performance optimization and some prior work has addressed it via offline analysis of memory traces. Wu et al. [121] proposed a data trace representation in a object-relative form and demonstrated its application in computing memory dependence frequencies and stride patterns. Our work is related to this work though we use object-relative traces in a different way and do not attempt to store or compress data traces. Zhong et al. [127] studied whole-program data locality pattern recognition. We adapt their technique in our framework to detect locality patterns at the object level.

Several studies have sought to improve data locality through compiler/OS interaction [16, 104, 19]. These studies focus more on avoidance of conflict misses, which is not a significant problem with L2/L3 caches nowadays because of their high degrees of associativity. In comparison, while sharing similar techniques such as page coloring with the above studies, we work on reduction of capacity misses of the shared cache. The problem is not much affected by the cache’s degree of associativity and we model the last-level cache as a fully associative cache.

In the field of embedded systems, cache partitioning has been studied to reduce cache misses and power consumption. In particular, a study by Ravindran et al. [99] shares some similarities with our work. However, our work differs from theirs in several aspects. First, our work is at the object level and requires no hardware support, while their work is at the instruction level and is based on cache way-partitioning. Second, we aim at reducing the number of L2 cache capacity misses instead of the number of L1 conflict misses. Third, our
framework does not use static analysis and no source level information is needed. Fourth, our goal is performance improvement instead of power reduction.

A very recent study by Soares et al. [106] also addresses the weakness of the LRU replacement policy with the last-level cache. It also uses page coloring as the basic support. However, our work differs significantly in several aspects. First, the approaches in the two studies are fundamentally different. Our proposed framework uses dynamic program analysis while Soares et al. focus on OS techniques. As a result, our technique explicitly uses object-level locality information, pays a one-time off-line profiling cost with small program inputs and thus has negligible run-time overheads. In comparison, the approach by Soares et al. collects data locality information at the page level and thus has a non-negligible on-line profiling overhead for each program execution. Second, our framework does not need any special architecture support, while the scheme of Soares et al. relies on a sampled-address data register (SDAR) only available on specific architectures such as recent PowerPC processors. Third, although both studies choose similar workloads, Soares et al. focus on sequential performance while our techniques are also applied to OpenMP benchmarks on shared-cache CMPs. Fourth, our approach supports arbitrary cache partitions while Soares et al. use a fixed-sized pollute buffer which may miss some optimization opportunities with large inputs.

5.10 Summary

We have designed and implemented a framework that partitions the last-level cache at the object level, in order to improve program performance for both single-thread and parallel data-sharing programs. The framework consists of several major steps including profile
generation, profile analysis and cache partitioning decision making and enforcement. Experimental results with benchmarks from a computational kernel suite, SPEC CPU2000 benchmarks and OpenMP NAS benchmarks demonstrate the effectiveness of our framework.
CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Contributions of the Dissertation

To address the challenges posed by modern and emerging architectures, the dissertation contributes to compiler optimization techniques through the application of data layout transformations for several important problems. In particular, the dissertation makes the following unique contributions.

Applying data layout optimization in compiling tensor contractions. Sequences of tensor contractions arise in many scientific applications based on ab initio electronic structure models in chemistry, physics and material science. These applications account for significant fractions of supercomputer usage at national supercomputing centers. We present an approach where a sequence of tensor contractions is implemented as a sequence of DGEMM calls interspersed with index permutation and redistribution to enable to use of the BLAS libraries and to improve overall performance. The costs of the constituent operations in the computation are empirically measured and are used to model the cost of the computation. This computational model is used to perform data layout optimization in
the synthesis of efficient sequential and parallel code that minimizes the overall execution time. Experimental results show the effectiveness of our approach.

**Optimizing the index permutation library.** Extensive research has been conducted on optimizing index permutation and related problems because of their ubiquitous usage and unique memory access patterns. We develop an approach employing both offline analysis and empirical search to decide optimization parameters for index permutation. We handle various alignments and conflict misses by generating multiple versions. Significant improvements are reported on different platforms with code generated by a special code generator. Several interesting observations demonstrate the effectiveness of our approach in exploiting hard-to-optimize architecture-specific features in the cache hierarchy and the memory subsystem. We believe our approach is very promising in optimizing other memory bandwidth-bound kernels and streaming applications.

**Optimizing data locality for NUCA chip multiprocessors at compile time.** CMP designs use a shared L2 cache to minimize the number of off-chip accesses. In order to handle increasing wire delays with diminutive feature sizes, shared L2 cache designs have employed a NUCA (Non-Uniform Cache Architecture), based on a banked organization with an on-chip network. The increasing wire delay makes exploitation of distance locality an important problem and it has been addressed by several hardware proposals. We develop a compile-time framework employing non-canonical layouts to localize L2 cache accesses for parallel programs, using a polyhedral model for program transformation. A set of optimizations are introduced to generate efficient parallel code. We have demonstrated significant improvements on a simulated 16-core chip multiprocessor.
Improving program performance through inter-object cache partitioning. Efficient on-chip cache utilization is critical to achieve high performance for many memory-intensive applications. While existing hardware and software solutions have focused on minimizing the cache contention between independent processes, inter-object cache contention has also posed a great challenge to optimization of both sequential and multi-threaded programs. To address this problem, we present a software framework for object-level cache partitioning that works across program inputs, cache configurations and compiler implementations. This framework consists of three components: 1) Profile generation: We collect object-level program profiles during training runs through binary instrumentation.; (2) Profile analysis: We determine objects’ locality patterns; (3) Partitioning decision making and enforcement: We properly allocate cache spaces to data objects aiming at maximizing cache usage. We have implemented the object-level cache partitioning support in the Linux kernel, and tested our framework on a commodity multi-core processor. Experimental results show the effectiveness of our system framework with single- and multi-threaded programs on several benchmarks including a computation kernel set, the SPEC CPU2000 suite and the NAS benchmarks.

6.2 Future Directions

We have demonstrated the effectiveness of data layout optimization in different contexts. We believe that data layout optimization in combination with computation transformations can also be applied to other problems. Below we discuss possible future directions for research.
Power consumption reduction through data layout optimization. Power has increasingly become a dominant constraint in both hardware and software designs. We can apply data layout optimization to reduce power consumption for emerging architectures.

- For manycore architectures, the network fabric is expected to account for a substantial portion of chip power consumption. We have shown in Chapter 4 that our proposed data locality optimization framework significantly reduces on-chip network link utilization and thus potentially reduces dynamic power dissipation. In comparison to previous proposals [103, 30, 74] on reducing energy consumption by the on-chip network, our framework has the unique advantage of not requiring special hardware support such as frequency scaling for individual channels. The power saving implication of our approach is worth further investigation with an accurate processor power model.

- Heterogeneous caches [114] can be created with different number of sets, associativities, threshold voltages, cell sizes, or even replacement and write-back policies. In order to improve energy efficiency, the inter-object cache partitioning framework can be extended to partition data among heterogeneous cache regions. For instance, data objects with little or no temporal reuse can be placed on a directly mapped cache region by the framework to save energy while maintaining the same overall performance.

Fine-grain inter-object cache partitioning. Many object-oriented applications such as server programs written in Java use small heap objects extensively. The object space of such an application can be divided into object sets with each set containing objects revealing a collective locality behavior. To improve program performance, we can include
small objects in the inter-object cache partitioning framework by facilitating memory co-allocation of objects from the same group. Finding accurate locality indicators for object groups, such as allocation sites and common containers, is critical to further investigation along this direction.
BIBLIOGRAPHY


[106] Livio Soares, David Tam, and Michael Stumm. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. In *MICRO ’08:


