

RICE UNIVERSITY

**Locality Transformations of Computation and Data for
Portable Performance**

by

Kamal Sharma

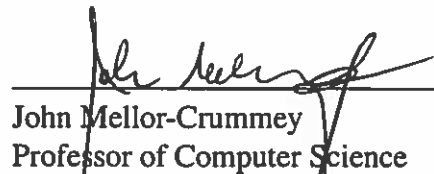
A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:



Vivek Sarkar, Chair
E.D. Butcher Chair in Engineering
Professor and Chair of Computer Science



John Mellor-Crummey
Professor of Computer Science



Timothy Warburton
Professor of Computational and Applied
Mathematics

Houston, Texas

August, 2014

ABSTRACT

Locality Transformations of Computation and Data for Portable Performance

by

Kamal Sharma

Recently, multi-cores chips have become omnipresent in computer systems ranging from high-end servers to mobile phones. A variety of multi-core architectures have been developed which vary in the number of cores on a single die, cache hierarchies within a chip and interconnect across chips. This diversity of architectures is likely to continue for the foreseeable future. With these architectural variations, performance tuning of an application becomes a major challenge for the programmer. Code optimizations developed for one architecture may have a different impact, even negative, across other architectures due to the differences in tuning parameters. This problem is compounded when scaling from a single node to multiple nodes in a cluster. Our thesis is that significant performance benefits can be obtained from locality transformations of computation and data that require minimal programmer effort. We establish this thesis by addressing three specific problems — *tile size selection*, *data layout optimization* and *automatic selection of distribution functions*. Tile size selection and data layout optimization improve intra-node performance, whereas automatic selection of distribution functions also enhances inter-node performance.

Loop tiling is a well-known locality optimization technique that can increase cache reuse at different levels of a memory hierarchy. Choosing tile sizes for different loop nests is a non-trivial task for a programmer due to the large parameter search space and different architectural parameters for different platforms. In this dissertation, we present analytical

bounds for tile size selection. Our approach uses different architectural parameters such as cache size and TLB size to limit the tile size search space, and also includes an automated algorithm to select optimized tile sizes.

Another important locality optimization is data layout transformation, which improves cache performance by restructuring program data. An important challenge for a programmer is to select an efficient data layout for a given application and target platform. In the past, selecting a data layout has been limited by the cost of rewriting applications for different layouts and by the effort required to find an optimized layout for a given platform. In this work, we present an automated tool for implementing different layouts for a given program. Using our approach, a programmer can experiment with different layouts across varying platforms for efficient performance. We also provide an automatic algorithm to select an optimized data layout for a given program and its representative executions.

When mapping an application onto multiple nodes, one challenge for scalable performance is how to distribute computation and data across the nodes. In this work, we introduce an algorithm to automatically select a task/data distribution function for a set of nodes. We build this solution on Intel CnC's distributed runtime system.

Overall, our approaches for the three problems provide automated methods for locality optimization that enable portable performance while requiring minimal programmer effort.

Acknowledgments

I would like to thank all members of the Habanero group at Rice University for their help during my thesis work. My dissertation has also built on discussions and collaborations with other researchers at Rice, Ohio State University, LLNL and Intel CnC team.

For my tile size selection topic, I would like to thank Jun Shirako at Rice and Prof. Sadayappan, Louis-Noel Pouchet and Naznin Fauzia from Ohio State University. Without their insightful discussions and feedback, tile size selection topic would likely have not been part of this dissertation. Their constant effort to overcome different challenges faced in this topic have improved my problem solving skills. Our findings have been published in a conference paper [1]*.

I would like to acknowledge help from James R. McGraw, Ian Karlin and Jeff Keasler from Lawrence Livermore National Laboratory (LLNL) for their help and support on the data layout optimization topic. Apart from their contributions to the technical work, they also made sure that my stay at LLNL was comfortable during my visits in 2011 and 2012. A technical report on this work has been published [2]*.

The distribution function topic was pursued in collaboration with Kath Knobe and Frank Schlimbach of the Intel Concurrent Collections (CnC) team. I am grateful for their help at various stages while pursuing this topic and providing me with various challenges during my internship at Intel in 2012.

I thank Prof. John Mellor-Crummey for agreeing to be a part of my thesis committee. His suggestions and feedback have always guided me during my PhD at Rice. I have thoroughly enjoyed deep technical discussions in his office. His constant effort to brainstorm the rationale behind the results and pursue new ideas have always motivated me. I have learnt a lot from him. I am also grateful to his HPCToolkit team for help investigating performance issues. In particular, I am thankful to Xu Liu from this team for collaborating with me on data layout optimization.

*This dissertation contains text from these papers and reports.

I am grateful to Prof. Keith Cooper for his support during my PhD. His stress relieving discussions have often worked magic for me. Sitting close to his office, he would pay a regular visit to my office especially when I was working. I have fun moments meeting him at different locations like the Rice gymnasium.

I would like to extend my sincere thanks to Prof. Timothy Warburton for agreeing to be a part of my thesis committee. His comments and feedback have been extremely useful in shaping this dissertation.

My thanks reach out to every member (present and past) of Habanero team for all the support during my stay at Rice. I am grateful to them for patiently interacting with me and providing me with valuable insights. I have also enjoyed attending the weekly Habanero meeting which has helped me improve my presentation skills.

Above all, I would like to thank my advisor Prof. Vivek Sarkar for helping me in all the projects. Without his support, most of this work would have not been possible. His guidance to pursue challenges in the right direction has helped me at various points during my PhD. I would also thank him for understanding my situation during my PhD and helping me during troubled times. His enthusiasm has led me to pursue most of the topics in this dissertation. I am grateful that he constantly believed in me.

A special thanks to my family for showing the necessary support throughout my PhD degree. My parents, Gopal and Vijaya Sharma, have always provided the necessary motivation and enthusiasm at various stages. Their constant eagerness to see their son walk the graduation ceremony, have always provided me the necessary impetus to pursue any upcoming challenge. I am also grateful to my wife, Manavi Sharma, for supporting me throughout this work. She has always been willing to serve late night food whenever needed. Without their balancing support, this work would have not been possible. I extend my gratitude to my aunts and uncles for their constant encouragement.

I am thankful to the Almighty to help me pursue every aspect of my PhD degree and making me complete this journey.

My research was partially supported by Defense Advanced Research Projects Agency

through AFRL Contract FA8650-09-C-7915, Lawrence Livermore National Labs through Contract B597790, Department of Energy through Contract DESC0008882, Intel through Contract CW1924113 and Texas Instrument Fellowship. I thank all these agencies for their generous grants.

Contents

Abstract	ii
List of Illustrations	xi
List of Tables	xv
1 Introduction	1
1.1 Thesis Statement	4
1.2 Thesis Organization	4
2 Background	6
2.1 Cache Structure	6
2.2 Translation Lookaside Buffer (TLB)	10
2.3 Locality Transformations	11
2.3.1 Tiling Transformation	13
2.3.2 Data Layout Transformation	14
2.4 Distributed Memory Systems	16
3 Tile Size Selection	19
3.1 A Motivating Example	24
3.2 Background	27
3.2.1 DL: Distinct Lines	28
3.2.2 Parametric Tiling	31
3.3 ML: Minimum Working Set Lines	31
3.3.1 Operational Definition of ML	32
3.3.2 Model of Computation	32

3.3.3	Distance in Tiled Iteration Space	33
3.3.4	Temporal and Spatial Reuse Distance	34
3.3.5	Definition of ML	39
3.3.6	Example	40
3.4	Bounding the Search Space by using DL and ML	41
3.4.1	Capacity Constraint for Intra-tile Reuse	42
3.4.2	Capacity Constraint for Inter-tile Reuse	42
3.4.3	Empirical Search within Bounded Search Space for Single-level Tiling	43
3.4.4	Compiler Pass for Bounded Search Space	44
3.5	Extension to Multi-level Tiling	45
3.5.1	Distance in Multi-Level Tiling	46
3.5.2	ML for Multi-Level Tiling	47
3.5.3	Bounded Search Space for Multi-level Tiling	47
3.5.4	Empirical Search within Bounded Search Space for Multi-level Tiling	48
3.6	Experimental Results	49
3.6.1	Performance Distribution of Different Tile Sizes	51
3.6.2	Search Space Reduction by DL-ML Model	53
3.6.3	Summary of Experiments	60
3.7	Summary	65
4	Data Layout Optimization	67
4.1	TALC Data Layout Framework	68
4.2	User Specified Layout Results	73
4.2.1	Test Codes	73
4.2.2	Experimental Methodology	75
4.2.3	Experimental Results	77

4.3	Automatic Data Layout Selection	92
4.3.1	Use Graph	92
4.3.2	Cache-Use Factor(CUF)	94
4.3.3	Automatic Data Layout Algorithm	95
4.4	Automatic Data Layout Results	97
4.5	Performance Anaylsis	99
4.5.1	Register Allocation Analysis	99
4.5.2	Locality and Prefetch Streams	100
4.6	Summary	101
5	Automatic Selection of Distribution Function	103
5.1	Intel CnC Programming Model	104
5.2	Distributed Cholesky Example	108
5.3	Distribution Function Selection Model	112
5.3.1	Framework for Parameter Generation	112
5.3.2	Parameter List	116
5.3.3	Overall Model using Linear Regression	118
5.4	Experimental Results	121
5.4.1	Performance Variation across Different Distribution Functions . . .	123
5.4.2	Linear Regression Model Results	124
5.5	Summary	125
6	Related Work	126
6.1	Tile Size Selection	126
6.2	Data Layout Optimization	129
6.3	Distribution Function Selection	132
7	Conclusions and Future Work	135

Bibliography

Illustrations

2.1	Cache configurations with various associativities. Each block represents a cache line.	7
2.2	An example three-level loop kernel	13
2.3	Tiled loop kernel. min denotes a minimum function comparison.	14
2.4	Candidate example for data layout transformation	15
2.5	Distributed memory system organization. NIC denotes network interface controller.	17
3.1	Matrix multiply IKJ loop order code	24
3.2	Normalized metrics for matrix multiplication with an IKJ loop	25
3.3	Address translation on Intel architectures	26
3.4	Sample code for explaining distance in tiled iteration space	34
3.5	Sample code 2 for explaining reuse distance.	36
3.6	Temporal reuse vectors for sample code 2	38
3.7	Matrix multiplication with single-level tiling	40
3.8	Search space for matrix multiplication for $T_1 = 30$	44
3.9	Compiler implementation of DL-ML bounds	45
3.10	Iteration space for level- k tile	47
3.11	Performance distribution for 3.11a matmult-3000x3000 and 3.11b 2d-jacobi-50x4000x4000 on <i>Nehalem</i> , <i>Xeon</i> , and <i>Power7</i>	51

3.12	Best tile sizes (achieves 95% or more of the performance with the optimal tile size) for matmult-3000x3000 with $k-i-j$ loop ordering on 3.12a <i>Nehalem</i> , 3.12b <i>Power7</i> and 3.12c <i>Xeon</i> . The x, y and z axes show tile size values for outer loop k , middle loop i and inner loop j respectively.	54
3.13	Best tile sizes (achieves 95% or more of the performance with the optimal tile size) for dsyrk-3000x3000 with $i-j-k$ loop ordering on 3.13a <i>Nehalem</i> , 3.13b <i>Power7</i> and 3.13c <i>Xeon</i> . The x, y and z axes show tile size values for outer loop i , middle loop j and inner loop k respectively.	55
3.14	Best tile sizes (achieves 95% or more of the performance with the optimal tile size) for 2d-jacobi-50x4000x4000 with $t-i-j$ loop ordering on 3.14a <i>Nehalem</i> , 3.14b <i>Power7</i> and 3.14c <i>Xeon</i> . The x, y and z axes show tile size values for outer loop t , middle loop i and inner loop j respectively.	56
3.15	Best tile sizes (achieves 95% or more of the performance with the optimal tile size) for dtrmm-3000x3000 with $i-j-k$ loop ordering on 3.15a <i>Nehalem</i> , 3.15b <i>Power7</i> and 3.15c <i>Xeon</i> . The x, y and z axes show tile size values for outer loop i , middle loop j and inner loop k respectively.	57
3.16	Best tile sizes (achieves 95% or more of the performance with the optimal tile size) for 2d-fdtd-100x2000x2000 with $t-i-j$ loop ordering on 3.16a <i>Nehalem</i> , 3.16b <i>Power7</i> and 3.16c <i>Xeon</i> . The x, y and z axes show tile size values for outer loop t , middle loop i and inner loop j respectively.	58
4.1	Extended TALC framework	69
4.2	Sample TALC field specification file	70
4.3	Sample TALC meta file	70
4.4	Sample C input file.	72
4.5	Stylized TALC output file.	73
4.6	IRSmk layouts selected for discussion.	77

4.7	IRSmk performance results on IBM Power 7 and AMD APU platforms with varying threads.	78
4.8	IRSmk performance results on Intel Sandy Bridge and IBM BG/Q platforms with varying threads.	79
4.9	IRSmk source code	80
4.10	SRAD layouts selected for discussion.	84
4.11	SRAD performance results on IBM Power 7 and AMD APU platforms with varying threads.	85
4.12	SRAD performance results on Intel Sandy Bridge and IBM BG/Q platforms with varying threads.	86
4.13	LULESH layouts selected for discussion.	88
4.14	LULESH performance results on IBM Power 7 and AMD APU platforms with varying threads.	89
4.15	LULESH performance results on Intel Sandy Bridge and IBM BG/Q platforms with varying threads.	90
4.16	Sample use graph	93
4.17	Hardware performance counter results for IRSmk on AMD APU	100
5.1	Matrix multiplication example in CnC. Note, this is a very simplistic example of matrix multiply in Intel CnC. Other implementations with finer granularity will be more efficient	104
5.2	Execution steps of Cholesky benchmark	109
5.3	Cholesky benchmark on distributed cluster nodes	111
5.4	Framework for parameter generation	113
5.5	Sample CnC Trace for Cholesky benchmark	114
5.6	Dynamic graph for Cholesky benchmark	115
5.7	Overall steps in Linear Regression Model for distribution function selection.	120

- 5.8 Cholesky performance variation across the different distribution functions. x-axis shows the sorted data points based on execution time. y-axis shows the percentage increase in execution time compared to the best point. Best point refers to the minimum execution time across the data points. 123

Tables

3.1	Cache characteristics of the architectures considered	50
3.2	Search space reduction factor across different architectures	60
3.3	1-level tiling results (time in seconds, N: Nehalem, P: Power7, X: Xeon) .	61
3.4	Empirical search results for 1-level tiling	62
3.5	2-level tiling results on <i>Xeon</i> – improvement over 1-level tiling	63
3.6	Parallel 1-level tiling results	65
4.1	Architecture and compiler specifications	75
4.2	Impact of source code lines changes across different layouts compared to base version. LOC denotes Lines of Code.	76
4.3	Speedup of best manual layout and automated layout speedup relative to base layout	97
4.4	Register spills for IRSmk on AMD APU for three different layouts	99
5.1	Different distribution functions. blk denotes blocks and numNodes represents number of cluster nodes in distributed environment.	122
5.2	Linear regression model results for Cholesky benchmark across all the data points.	124

Chapter 1

Introduction

The multi-core revolution has resulted in a paradigm shift from single-core to multi-core processors. This shift occurred not only in desktops and servers but also in tablets and mobile phones. This range of platforms employ multi-core chips with a range of architectures and capabilities. For example, multi-core processors on servers are more performance oriented, whereas on mobile devices, they are more focused on energy efficiency. Performance optimization becomes an ever increasing challenge due to the complexity of these architectures. The performance optimization problem scales out even further as we move from single node to multi-node configurations such as server/cluster systems. To address this challenge, programmers generally adopt two approaches. First, programmers rely on back-end automated compiler systems to extract performance from a given target architecture. But, a key limitation of automatic compiler optimizations is that it can take at least two to three years to develop a high-quality compiler for a given architecture [3]. In the second approach, a programmer may spend significant amount of time, perhaps days or weeks, to manually tune an application for a given single/multi-node platform. However, a key limitation of this approach is that programmer's optimization of an application is often not portable across machines. In this work, we develop three approaches — tile size selection, data layout optimization and automatic selection of distribution functions, to enhance

portable performance optimizations across different machines. The first two approaches are locality optimization techniques which improve cache use efficiency to optimize overall performance within a node. The third approach models addresses distributing of data and tasks across nodes in a multi-node environment. Note that portable performance refers to transformations that are portable across different machines by automatically exploiting the underlying machine characteristics.

Loop tiling is an important locality optimization techniques that reorganizes the computation iteration space to improve cache reuse, thereby reducing accesses to memory. By reusing data, loop tiling helps in reducing expensive cache misses and thereby improving the performance of an application. Loop tiling has been widely studied in the past [4, 5, 6, 7, 8, 9]. From this point onward, we will use the terms “loop tiling” and “tiling” interchangeably. Tile size is one of the key parameters in tiling as it directly impacts the amount of reuse in a cache. Selecting a large tile size will result in data being displaced from cache, whereas a smaller tile size reduces the amount of reuse in cache; both extremes can have sub-optimal performance. Moreover, tile size depends on architectural features such as levels of memory hierarchy, cache line size, cache size, TLB line size and TLB size. Thus, selecting tile sizes becomes a major challenge for any system, whether it is done by a compiler or a programmer. In this work, we develop analytical bounds for tile size selection [1]. Our techniques uses architectural features such as cache size and cache line width to automatically construct the bounds for a given loop nest. Using our approach, an automated system or programmer can limit the tile sizes considered to a bounded space rather

than exploring all the possibilities. Our experimental results show search space reduction factors of upto $11,879\times$ for tiling on modern processors such as *Intel Xeon*. Results also indicate that automated tile size search techniques such as Nelder-Mead Simplex [10] and Parallel Rank Ordering [11] can be improved using our analytical bounds.

Another important locality optimization is data layout transformation. Data layout transformations improves cache performance by restructuring data in the programs which helps in better packing of data elements for cache reuse, e.g., converting a struct of arrays to an array of structs or changing the layout of a multidimensional array by using space-filling curves. Past research has demonstrated that data layouts can have a significant performance impact of an application [12, 13, 14, 15]. However, these optimizations have not been widely adopted due to two reasons. First, selecting a good data layout is a big challenge for a given architecture. Second, there is a huge cost to re-write applications to use different layouts across varying architectures. In this dissertation, we demonstrate a source-to-source data layout optimization, which performs different array interleavings based on a layout specification [2]. Using our technique, a programmer can automatically adapt his application across different layouts without re-writing them. Our experimental results show performance improvements upto $20\times$ on *IBM Power7* for the *IRSmk* benchmark. We also develop a new approach to automatically recommend a good layout for a given source program using machine characteristics such as cache line size.

When scaling out from single node to multi-node environments, performance challenges become compounded due to increased system complexity. A programmer has to not

only focus on optimizing their program on a single node, but also reason about effects of communication and task dependencies in an inter-node environment. Distributing task and data across nodes is a major challenge in such distributed configurations. In this dissertation, we develop an automated model to efficiently select a distribution function, which maps task/data across nodes, for a given application. We use the Intel CnC programming model [16, 17] to demonstrate our approach. Using our method, a programmer can select a good distribution function, without exploring the entire search space for distribution functions.

1.1 Thesis Statement

Our thesis is that significant performance benefits can be obtained from locality transformations of computation and data that require minimal programmer effort. We establish this thesis by exploring locality transformations of computation and data in three well-known problems: tile size bounds for loop tiling, data layout optimization and selection of inter-node task/data distributions. Using our proposed approaches, a programmer would not only save tuning time for a given platform, but also improve overall performance across different platforms.

1.2 Thesis Organization

The rest of this thesis is organized as follows:

- Chapter 2 contains background information on memory hierarchies in modern processors.
- Chapter 3 describes our results for tile size selection.
- Chapter 4 describes our results for data layout optimization.
- Chapter 5 describes our results for automatic selection of distribution function.
- Chapter 6 includes a comparison with past related work.
- Chapter 7 presents our conclusions and also discusses some potential topics for future work.

Chapter 2

Background

The chapter describes background material presented in this dissertation. First, we introduce cache structure present in processor architectures. Second, we present concepts of Translation Lookaside Buffer (TLB). Then, we describe various locality transformations that form the foundation for our work. Finally, we discuss concepts of distributed memory systems. All of these topics help in understanding the different methods developed in this dissertation. In this section, we have limited the scope of background information to relevant terms used in the later chapters.

2.1 Cache Structure

Cache is an associative small memory present between the processor and main memory to reduce the memory latency by capturing frequently accessed data [18, 19, 20]. On Intel Nehalem systems, read access latency to cache closest to the processor (L1) is 1.3 ns whereas to memory is 65.1-106 ns [21]. This latency difference leads to huge performance gains when values are present in the cache memory.

Due to access patterns within a program, caches benefit from locality of element values that are frequently accesses across iterations. There are two types of locality present in a

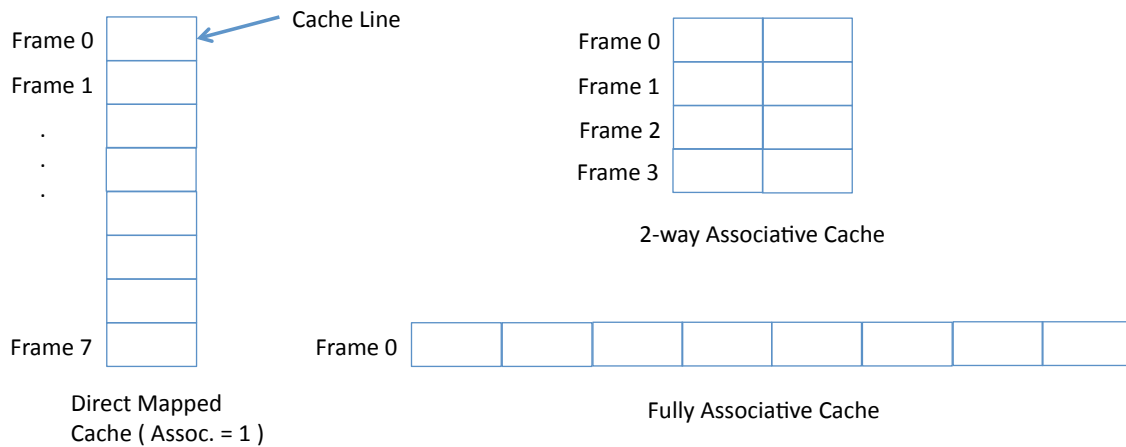


Figure 2.1 : Cache configurations with various associativities. Each block represents a cache line.

program : *temporal locality* and *spatial locality*. *Temporal locality* refers to the property where a current value is likely to be accessed in the future. Thus, holding these values within a cache will benefit future accesses. *Spatial locality* denotes that nearby addresses values to the current value will be referenced in the future. Thus, fetching contiguous values from memory and storing them inside the cache will reduce the latency for future accesses. For example, accesses to $A[i+1]$ is very likely whenever the program accesses $A[i]$. However, both of these locality types is limited by the cache parameters.

Three parameters affect cache organizations – cache size, line size and associativity. Cache size is the overall capacity of the cache and is measured in bytes. When a cache reaches its capacity, it employs a replacement algorithm such as least recently used (LRU) for the new value to be stored in the cache by evicting the old values. If the cache size is small, often for level 1 caches, values might be frequently flushed out leading to no

reuse for future accesses to same cache line and imposing high latency accesses to main memory. Line size is the unit of transfer between main memory and cache [18]. Line size determines the number of data values can be held together inside the cache. For example, if line size can hold four values, then a data transfer would bring “A[0]...A[3]” values together from main memory whenever any one of them are accessed, assuming A[0] is aligned to a cache line boundary. Associativity refers to degree of freedom to place a given cache line within a cache. Whenever, there is a cache lookup to determine if a cache line is present or not, it is mapped to corresponding cache frame. In this section, cache frame and cache set are identical. A cache frame may hold one or more lines, depending upon associativity, needs further examination to determine the value’s presence. The replacement algorithm will determine which line should be replaced in case the value is absent. If the associativity is 1 i.e. a single cache line maps to each cache frame, then it is referred to as direct mapped cache. However, if a cache line can be present in any of the cache frames, such cache is called as fully associative cache. On most systems, associativity usually ranges from 2-32. Figure 2.1 shows the different cache configurations with varying associativity.

To mitigate the effects of conflicts misses arising due to low associative caches, Jouppi [22] proposed victim cache. Victim cache is a fully associative cache present between the first level and next level of memory. As this cache is fully associative, it does not suffer from conflicts arising from same cache line being mapped to the same frame/set. The original proposal of victim cache was using direct mapped cache. However, modern architectures such as IBM Power 7 have a large victim cache at L3 level [23].

As the cache capacity becomes a constraint, modern computer architectures employ multiple levels of cache as opposed to a single cache. For example, Intel Nehalem has three levels of cache [24]. The level of cache determining the closeness to the processor, starting from one. Caches closer to the processor are smaller and more expensive as compared to caches closer to main memory. There is an increase in the cache access latency as we move away from level 1 (L1) cache. Often on modern multi-core architectures, caches beyond L1 are shared across two or more cores to enhance reuse between the cores.

Multi-level caches enable another design choice for chip manufacturers, whether to hold data values in any one level of cache (exclusive cache) or compose lower levels of cache as subset of values from higher levels (inclusive cache). Exclusive caches store data at any level but not multiple levels of cache. This property helps increase the cache capacity as data present in L1 is separate from data present in L2 (level 2) and avoids duplication. On a cache miss, data is brought from the level it is found and exchanged with cache line at those level. A disadvantage of exclusive cache is data must be checked at all levels of cache to invalidate a cache line. Inclusive cache have the property that data in lower levels must be present in higher levels of cache, where levels increase as we move away from the processor. Thus, data present in L1 is always present in L2. On a cache miss, cache line is fetched from higher levels of cache if it is present there. An obvious disadvantage of this approach is to duplication of data across different levels of cache. An advantage of this approach is that cache invalidations by other processors are simplified as a check for cache line presence is only performed at the highest level of a processor cache.

Another important classification for caches depends on the values they hold. Caches can be classified as data cache, instruction cache or unified cache. As terms suggest, data cache stores frequently accessed data values in a program whereas instruction cache stores the instructions of a program. Unified cache stores both data and instructions. In this dissertation, we focus on improving data cache effects as most of the benchmarks suffered performance loss due to data cache limitations. We observed less profound effects of instruction caches for the benchmarks used in this work.

2.2 Translation Lookaside Buffer (TLB)

In the earlier section, we discussed data and instruction caches. There is another cache used for storing the virtual to physical page translations called as *Translation Lookaside Buffer (TLB)*. TLB's have long existed on processors as far back as the IBM 360 processor introduced in 1964 [25]. TLB is essential to a processor since programs access data via virtual memory address and a cache is needed to store the mapping of these addresses to physical memory address. Without a TLB, every memory access would impose a high overhead to translate virtual address to a physical address. TLB's act as a cache to store recently performed translations for better performance of an application. TLB misses are more expensive than data cache misses due to translation overheads. Even simple kernels such as matrix multiplication have improved performance by optimizing TLB performance [26]. However, the overall organization of TLB is different from data caches.

In comparison with data caches, TLB store page entries as a cache frame. Thus, TLB

line size is equivalent to a page size which is quite large when compared to data cache line size. On Intel Nehalem, TLB page size is configurable to 4KB, 2MB or 4MB [27]. Similar to data caches, TLB's are also organized in multiple levels, often two or three. Page not found in the first level TLB is looked up in subsequent levels. Access latency increases as we move from one level to the other. If the page is not found in any of the TLB's, a TLB miss occurs and processor does a page walk to find the appropriate page entry and bring it into the TLB. Older pages are retired from TLBs by using replacement algorithm such as LRU.

Chip manufacturers may provide separate specialized hardware to speedup the page translations. For example, IBM Power 7 uses Effective to Real Address Translation (ERAT) cache which enables direct translation of effective (software address) to real addresses (physical memory addresses) [28]. ERAT is a first-level and fully associative translation cache on Power 7, which helps in improving the overall performance of page translations.

2.3 Locality Transformations

Caches and TLB help in caching frequently accessed data in a program. This behavior prevents expensive memory access for each data access, which often costs hundreds of clock cycles. The discrepancy between processor frequency and memory access latency have existed for more than two decades now, known as “memory wall” problem [29]. Programs usually face huge stalls due to memory accesses which pose high latency. To overcome the memory wall problem, locality transformations are applied to programs to

improve their overall performance. Locality transformations exploit the spatial and temporal reuse of accesses in a program. Such transformations are generally carried out by automated compiler passes or by an expert programmer. Locality transformations enable better use of memory hierarchy in an architecture, leading to more cache hits and improved performance gains. Note that transformations must be done in a way so that they do not change the program semantics.

In the past work, locality transformations have been classified into two categories: reorganize instructions and reorganize data. These two categories can be used separately or in conjunction with each other to promote better cache reuse. To reorganize instructions, compilers mainly focus on loops in a program. Various loop transformations such as tiling, fusion, distribution, skewing and reversal have been proposed in the literature to exploit better locality [30,31,32,33]. In this dissertation, we focus on a sub-problem in tiling. For data reorganization, compilers transform the data, i.e. arrays present in a program, without changing much of the program structure. Due to this data transformation, the access pattern to the arrays present in the memory changes, possibly, leading to better cache reuse. In this dissertation, we develop array regrouping as part of data transformation. However, loop transformations and data transformation are not mutually exclusive transformations and can be combined together to have a better overall cache reuse. This dissertation presents new approaches for sub-problems in these transformations separately.

2.3.1 Tiling Transformation

Tiling, also known as blocking, is a loop transformation which improves data locality of a program by reusing elements in the cache [5,30]. Tiling transformation divides the loop iteration space into smaller chunks, so that data elements accessed in a loop are not evicted out of the cache. Using this approach, data elements that have smaller reuse distance are directly accessed from the cache. Different levels of memory hierarchy can benefit from tiling including TLBs and cache simultaneously. Thereby, tiling based on loop access patterns often improves the overall execution time. Tiling has shown to benefit numerical scientific applications by improving locality across loop kernels [34,35].

Tiling is a combination of two other well-known loop transformations, strip-mining and interchange. Thus, the legality of tiling transformation depends on the strip-mining and loop interchange. From dependence theory, we always know that strip-mining is a legal transformation for a given loop. Now, tiling legality depends on legal conditions for loop interchange. From dependence theory, interchange is legal if the direction is “=” or “<” for dependence carried by any loops in a kernel [36].

```

for (k = 1 ; k < N ; k++ )
  for (i = 1 ; i < N ; i++ )
    for (j = 1 ; j < N ; j++)
      ... = A[i][j] + ...;

```

Figure 2.2 : An example three-level loop kernel

Figure 2.2 shows a simple three-level loop kernel with A array access. Figure 2.3 shows the corresponding tiled version of the same loop kernel. From these two figures, we clearly

```

for (ii = 1 ; ii < N ; ii+=Ti )
  for (jj = 1 ; jj < N ; jj+=Tj )
    for (k = 1 ; k < N ; k++ )
      for (i = ii ; i < min(ii+Ti,N) ; i++ )
        for (j = jj ; j < min(jj+Tj,N) ; j++ )
          ... = A[i][j] + ...;

```

Figure 2.3 : Tiled loop kernel. min denotes a minimum function comparison.

see that the original three-level loop kernel is transformed into five-level loop kernel by tiling. In the original case, assuming N is large, all the elements of A array are accessed before running the next iteration of k -loop. Thus, the cache will be unable to hold any array elements, leading to poor data locality. However, the tiled loop chunks the iterations of i and j loops. This approach enables to revisit elements of A array in cache more frequently as compared to the original loop. The value of T_i and T_j parameters determine the number of elements to be kept in the cache as visited in k -loop. In Chapter 3, we will have more in-depth discussion about selecting the right parameters for loop tiling.

2.3.2 Data Layout Transformation

Locality optimization improves the performance of a program. In Section 2.3.1, we have discussed how tiling reorders the iteration space to help improve locality of programs. Besides reordering loops, there is another way to improve locality by means of data layout transformation. Data layout transformation changes the organization of data in such a way that subsequent data accesses are found in cache. Improving cache behavior leads to overall improvement in program execution by proper packing of data for spatial and

temporal reuse. Scientific applications have significant amounts of data array accesses. Array accesses may be improved by changing their data layouts.

```

for ( i = 1 ; i < N ; i++ )
    for ( j = 1 ; j < N ; j++ )
        C[i][j] = A[i][j] + B[j][i];

```

Figure 2.4 : Candidate example for data layout transformation

Figure 2.4 shows a simple candidate example for data layout transformation. Assuming that the program has row-major access order, arrays C and A have efficient cache behavior due to spatial locality. However, B array has poor locality since each array access would fetch a different cache line as it has column access order. Interchanging the loop from i-j to j-i would benefit B array, but would lead to poor locality for C and A. A simple method to improve the overall performance is to change the data layout of B. Instead of storing data in the default row-major order, an efficient compiler would apply data layout transformation, so that the array is stored in column contiguous order. Before transformation, the data layout for array B would be B[0][0], B[0][1], B[0][2] ... B[1][0], B[1][1] ... ,B[1][N] ... B[N][N] and after transformation, it would be B[0][0], B[1][0], B[2][0] ..., B[N][0], B[0][1], B[1][1] ..., B[N][1], ... B[N][N]. The compiler must also make sure that legality conditions for this transformation are valid across the program. In this case, compiler must update all the accesses to B array across the program to ensure correctness. An alternative approach that the compiler may employ is to transform an array just before the loop and re-transform the array after the loop. Often, such operations are quite expensive in terms of program

execution and lead to poor performance when there are multiple small loop kernels.

The above example shows one kind of data layout transformation. There are many other data layout transformations, including Array Padding [37], Array Regrouping [15] and Array alignment [38]. In Chapter 4, we will focus on a novel framework for data layout optimization.

2.4 Distributed Memory Systems

Distributed Memory Systems employ multiple shared memory systems (or single-node machines) for faster execution of programs. Different challenges emerge as programs scale from a shared memory system to a distributed memory system. In shared memory systems, programmers tend to focus on optimizing their program without focusing data address space, since data is accessed locally. However, as programs scale-out to distributed systems, programs need to appropriately fetch remote data before any computation can be performed on them, due to different address space.

Figure 2.5 shows a distributed memory system organization. In this organization, multiple shared memory systems are connected through an interconnect network. Each shared memory node contains multiple processors and has deep memory hierarchy which contains multi-level caches (shared and exclusive cache) and main memory. Tasks running on a processor can access all the local available data on a shared memory system. The interconnect network plays an important role in communicating data between nodes. This network can be organized as point-to-point links or sharing common links. There are var-

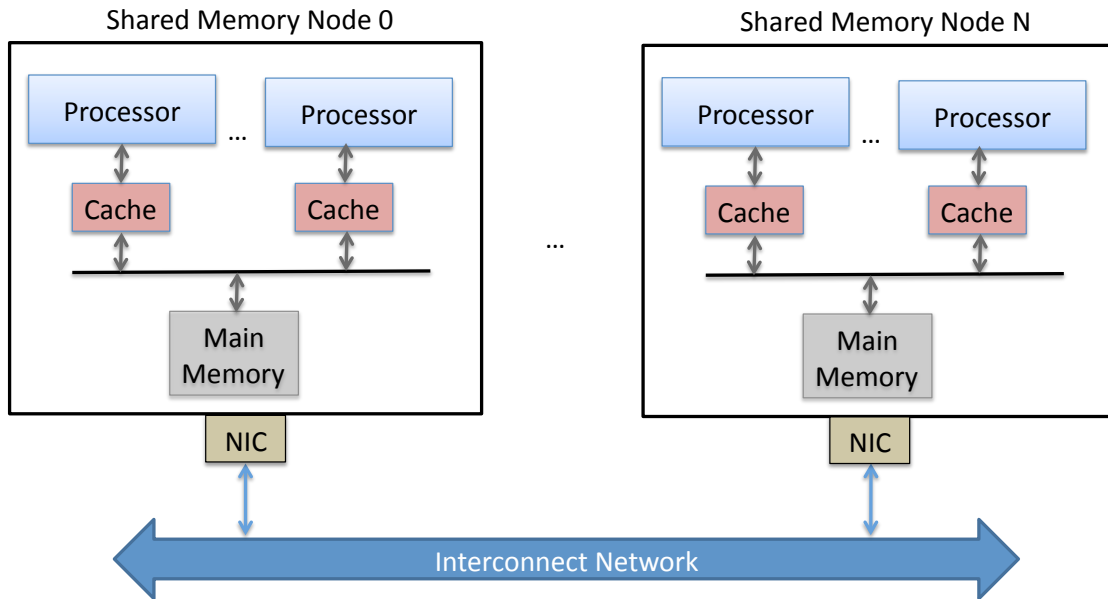


Figure 2.5 : Distributed memory system organization. NIC denotes network interface controller.

ious network topologies such as bus network, ring network and torus interconnect which determines the communication scalability of distributed systems. Whenever any data needs to be fetched from a remote system, a communication is initiated through the interconnect network.

Two key challenges that arise while programming for distributed systems are: distributing task and data across the nodes and communication between nodes. Proper distribution of task and data is important for load balancing the entire system, whereas communication is important as remote data fetches are more expensive in terms of application time. There have been various programming models such as MPI [39], OpenMPI [40], UPC [41] and Intel CnC [16], which aid in programming distributed systems. Some of these program-

ming models such as MPI provide explicit control for communication, task/data control and synchronization; other models such as Intel CnC uses a runtime to manage these activities. Nevertheless, tuning applications to scale across distributed systems is a non-trivial problem. In Chapter 5, we present our approach to tune applications on distributed systems.

Chapter 3

Tile Size Selection

Modern computer systems employ multi-level memory hierarchies, with the latency of data access from higher levels of memory hierarchy being orders of magnitude higher than the time required to perform arithmetic operations. Reduction of data access overheads is therefore critical to achieving high performance. *Tiling* [4,5,6,7,8,9] is a classical technique to enhance data reuse in fast caches for loop-oriented computations. Recent advances have resulted in the development of software to automatically generate parametrically tiled code, even for imperfectly nested loops [42, 43, 44, 45]. It is well known that the choice of tile sizes has a significant effect on performance, but *effective selection of optimized tile sizes* remains an open problem that has become more challenging as processor memory hierarchies increase in complexity.

The two main types of approaches pursued for tile size selection in past work are *analytical* and *empirical*. In analytical approaches, a compiler selects tile sizes based on static analysis of loop nests and known characteristics of the memory hierarchy. Although several analytical techniques have been proposed in the literature for tile size optimization [46, 47, 48, 49, 50, 51, 52, 53, 54], none has been demonstrated to be sufficiently general and robust to be used in practice for selecting the best tile sizes. It is unrealistic to build a purely analytical model for tile size optimization that is capable of accurately modeling

the complex characteristics of modern processors with multiple levels of parallelism and deep memory hierarchies. As a result, the gap between the performance delivered by the best known tile sizes and those selected by an analytical approach has continued to widen, thereby diminishing the utility of past analytical approaches.

In contrast, empirical approaches to tile size optimization treat the loop nest as a black box, and perform empirical *auto-tuning* for a given architecture [55,56,57,58]. The highly successful ATLAS (Automatically Tuned Linear Algebra Software) system [55] uses empirical tuning at library installation time to find the best tile sizes for different problem sizes on the target machine. One of the most challenging issues in empirical approaches for tile size selection is the huge search space that needs to be explored when tiling multiple loops. Further, the increasing depth of memory hierarchies in modern computer architectures fundamentally requires multiple levels of tiling [59], which makes the search space impractically large. In the near future, there would be an increase in the complexity of choosing efficient tile sizes due to growth in memory hierarchy levels. As a result, most empirical tuning frameworks employ simplifications to reduce the tile size search space.

One major simplification is to consider only “square” tiles, i.e., equal tile sizes along all dimensions. For example, ATLAS ver3.9.35 uses single-level square tile search optimized for the L1 cache. It is obvious that the square-tile approach drastically shrinks the search space by the order of N^{l-1} where l represents the loop nest level. However, considering non-square tiles is important since it has been shown for many domains (e.g., by Goto [26] for linear algebra and by Datta et al. [60] for stencil codes) that optimal tile sizes are

unequal in the different dimensions. The experimental results in this paper support this observation with performance speedups of up to $1.40\times$, $1.28\times$, and $1.19\times$ for the best non-square tile sizes relative to the best square tile sizes on three different architectures (Xeon, Nehalem, and Power7 respectively).

Another approach reducing the tile size search space is to use analytical approaches to find candidates for empirical search [61, 62, 63]. A significant work was introduced by Chen et al [62], which employs standard compiler heuristics to find profitable candidates of optimization variants including loop permutation order, unroll-and-jam and single-level tiling, although the optimal point, e.g., the best tile size in tiling, is not guaranteed to be found by existing compiler analyses.

Since the search spaces for tile size selection increase explosively for multidimensional non-square tiles and even more for multi-level tiling, an effective approach to prune the search space is essential for any search strategy to find the best tile size. Furthermore, while expensive empirical tuning is feasible for libraries such as BLAS that are tuned once per machine and reused across applications, tiled user codes usually require the empirical search to be done in a reasonable amount of time since it needs to be performed on all the time-consuming loop nests in the applications.

Here, we introduce novel analytical bounds for empirical tuning of non-square multi-level tiling. The analytical bounds help in vastly reducing the search space to a bounding region which contains a near optimal tile size. The proposed approach to pruning the search space is complementary to and can be combined with any existing empirical search

strategies, e.g., the analytical bounds can be integrated with existing auto-tuning frameworks such as ATLAS [55]. The experimental results show that our approach can reduce the search space by up to four orders of magnitude. The reduction factors ranged up to $81\times$ - $11880\times$, $21.90\times$ - $1978\times$, and $1.46\times$ - $1142\times$ on the Xeon, Nehalem, and Power7 respectively, for five benchmarks (matmult, syrkc, trmm, 2d-jacobi, 2d-fdtd) that we studied. The proposed approach considers both sequential and parallel (shared-memory) tiled programs, and the experiments indicate that optimal tile sizes for both sequential and parallel codes can be found in the proposed boundaries.

The proposed approach employs a pair of analytical models to prune the search space — a conservative model that under-estimates the tile size (DL), and an optimistic model that over-estimates the tile size (ML). A conservative model from past work, DL (Distinct Lines) [64], models the required cache capacity for a tile as its total data footprint. Thus, with DL, we assume distinct space in cache for every referenced data element, regardless of the order in which data elements are accessed within a tile. This is a pessimistic assumption for many applications, especially applications with streaming data accesses. We also introduce an optimistic analytical model, ML (Minimum Lines), that assumes ideal intra-tile cache block replacement. With the optimistic ML model, the cache usage estimated for a given tile volume never exceeds that of the conservative DL model. Conversely, for a given cache size, the tile volume (number of iterations) allowed by the ML model is larger than that of the DL model. Thus, DL and ML respectively provide lower and upper bounds for tile sizes. We use the aggressive ML model and the conservative DL model to bound the

tile size search space for empirical tuning, and show in our experiments that this bounded sub-space still contains optimal tile sizes, despite reductions of up to 4 orders of magnitude in the size of the search space.

Our main contribution is to propose analytical models to bound the space of candidate tile sizes, that takes into account multi-level data caches and TLBs. The results in Section 3.6 reinforce the following points:

1. Our analytical models significantly reduced the search space size, while preserving the optimal points in the pruned space.
2. In most cases, the optimal points have non-square tile sizes with performance improvements of up to 40% improvements over square tiles.
3. The proposed model is also effective in finding optimized tile sizes for parallelized code.

This chapter is organized as follows. Section 3.1 reinforces the motivation for this work via a case study that showcases some of the challenges arising from modern memory hierarchies. In Section 3.2, we provide background on parametric tiling and the DL model from past work. Section 3.3 introduces the new ML model for single-level tiling, and Sections 3.4 elaborates on how the DL and ML models can be used to bound the search space for empirical tuning. Section 3.5 extends the ML model to multi-level tiling. Section 3.6 presents experimental results on three platforms using a number of benchmarks, demonstrating the effectiveness of the approach. Optimal tile sizes were always found within the

```

// inter-tile loops
for ii = 1 to N, Ti
  for kk = 1 to N, Tk
    for jj = 1 to N, Tj
      // intra-tile loops
      for i = ii to min(ii+Ti,N)
        for k = kk to min(kk+Tk,N)
          for j = jj to min(jj+Tj,N)
            C[i][j] += A[i][k]*B[k][j];

```

Figure 3.1 : Matrix multiply IKJ loop order code

reduced search space.

3.1 A Motivating Example

Past work on performance models for tile size selection were developed for minimizing first level capacity and conflict cache misses [30, 46, 47]. In this section, we illustrate the impact of higher levels of data cache and Translation Lookaside Buffer (TLB) on tile size selection. As a motivating example we provide a detailed analysis of the execution of a tiled Matrix-Multiply kernel, an example that has been studied extensively. Figure 3.1 shows a sample code from [46], with the the IKJ loop order.

Tiling is a critical transformation to increase data locality in this case: T_i , T_j and T_k must be selected such that the data accessed during the computation of a tile fits entirely in the first cache level, to avoid costly capacity misses.

Reuse analysis that considers only L1 cache suggests that we set T_i to N , in order to take full advantage of temporal data locality for the matrix B along the i loop [46]. This solution is motivated by the fact that no element of B will be used in two different

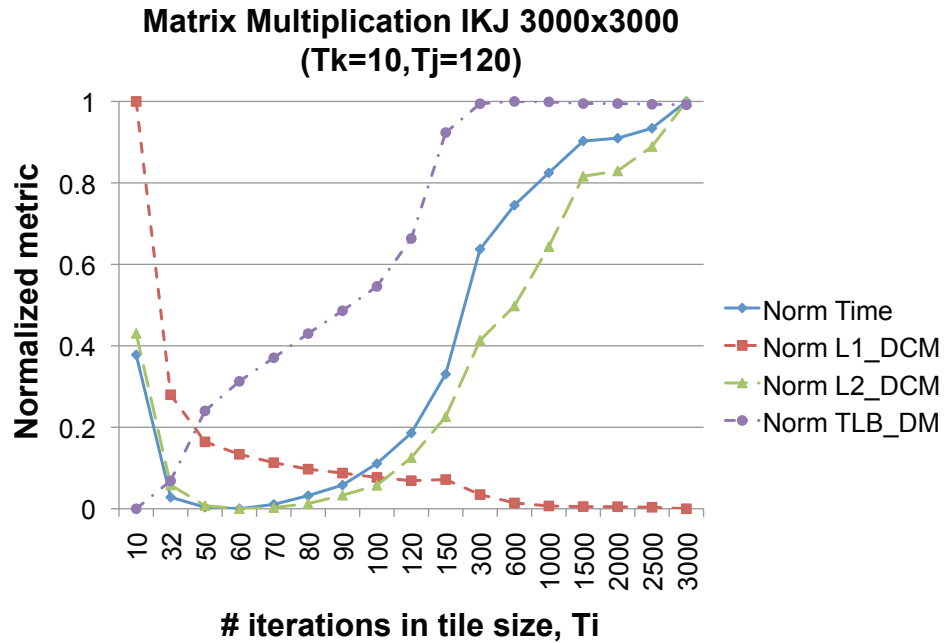


Figure 3.2 : Normalized metrics for matrix multiplication with an IKJ loop

tiles, an apparently ideal solution in terms of L1 cache misses — provided T_j and T_k are selected adequately. Furthermore, setting $T_i = N$ allows us to explore a two-dimensional search space $T_k \times T_j$ instead of a three-dimensional search space $T_i \times T_k \times T_j$, thereby significantly reducing the search space to look for optimal tile sizes.

This solution focuses only on minimizing L1 cache misses. To illustrate the deficiency of a Level 1 cache-centric approach, we now report a detailed analysis for an Intel *Xeon* (E7330) 2.40 GHz processor with 32KB L1 cache, 3MB L2 cache, 16 Entries TLB1 and 256 Entries TLB2 (4KB Page Size) for a problem size of 3000×3000 . After performing an exhaustive empirical search for the tile sizes, we found that the optimal tile size on this machine is $(T_i, T_k, T_j) = (60, 10, 120)$. Note that this optimal point has unequal tile sizes in

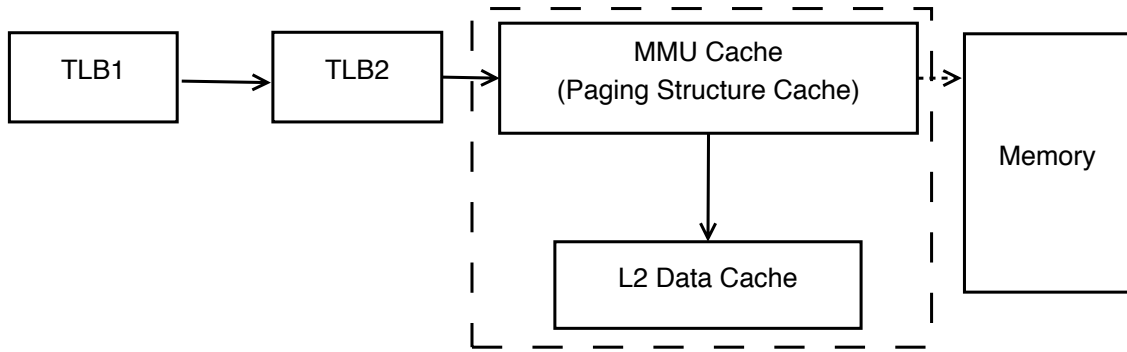


Figure 3.3 : Address translation on Intel architectures

different dimensions because each array dimension has different data reuse distance, and efficient vectorization needs the vectorized dimension (innermost tile size T_j) to be large. To illustrate the performance impact of the values of T_i , we fix $T_k = 10$ and $T_j = 120$, and plot the various values for T_i in Figure 3.2. This graph plots *normalized values* for a number of metrics i.e., the y-axis plots the ratio of the metric for the optimized case relative to the unoptimized case. The metrics plotted in Figure 3.2 are: L1 Misses (PAPI.L1_DCM : L1D_REPL)*, L2 Misses (PAPI.L2_DCM : L2.LINES_IN)[†], TLB2 Misses (PAPI.TLB_DM : DTLB.MISSES)[‡] and execution time, obtained by varying T_i .

First, we observe that L1 misses decrease as T_i increases, as expected. However, from the graph it is clear that the optimal tile size does not occur at $T_i = 3000$. At $T_i > 90$, we see that the execution time shows an upward trend, in contrast to what the L1 cache consideration suggests. As the cache footprint of the kernel increases, it results in larger

*All performance counters were collected using standard PAPI interface. PAPI.L1_DCM: L1D_REPL - Number of lines brought into L1 cache.

[†]PAPI.L2_DCM: L2.LINES_IN - Total cache lines allocated in L2 cache.

[‡]PAPI.TLB_DM: DTLB.MISSES - Level 2 TLB misses.

TLB2 misses and L2 misses, eventually leading to substantial degradation in execution time.

To better understand this effect, Figure 3.3 shows the virtual to physical address translation for this machine. More details on virtual address translation can be found in [65,66]. A lookup for address translation is performed at both the TLB levels. When an entry is not found in both TLBs, a radix tree page walk is performed, with higher (leftmost) bits fetched from the MMU cache and lower (rightmost) bits from the L2 data cache. When address translation bits are not found in L2 cache, it causes DRAM accesses, leading to significant stalls for an application. Thus, increasing an application’s footprint causes significant pressure on data cache and TLB. This page walk behavior is not just restricted to Intel architectures. AMD’s Page Walking Cache and PowerPC’s Hashed Table approach exhibit a similar characteristic, where address translation requires traversal of data caches in case of TLB misses [65,67].

These observations led us to rethink the tile size selection model. While it is still critical to optimize for reuse in the L1 data cache, one also needs to consider the reuse effect at Level 2 of the memory hierarchy even for single level tiling. This leads to considering the $T_i \times T_k \times T_j$ space to find the actual optimal tile size.

3.2 Background

In this section, we introduce Distinct Lines (DL) model, which had been developed in the past for tile size selection. This model was developed by Sarkar et al. [53,64]. Then,

we provide a brief overview of parametric tiling, which explores the current tile size search space.

3.2.1 DL: Distinct Lines

The DL (Distinct Lines) model was designed to estimate the number of distinct cache lines accessed in a loop-nest [53, 64]. Consider a reference to an m -dimensional array variable (called A , say), enclosed in n perfectly nested loops with index variables i_1, \dots, i_n :

$$A(f_1(i_1, \dots, i_n), \dots, f_m(i_1, \dots, i_n)) \quad (\text{Fortran})$$

$$A[f_m(i_1, \dots, i_n)] \cdots [f_1(i_1, \dots, i_n)] \quad (C)$$

where f_j is an affine function of i_1, i_2, \dots, i_n . Let t_k denote the tile size of loop- i_k ($1 \leq k \leq n$) when rectangular parametric loop tiling is applied to the n -th perfectly nested loops. We compute the number of distinct cache lines accessed in a tile, $DL(t_1, t_2, \dots, t_n)$, in the following four steps [53].

1. Computing array subscript range for each dimension:

$range_j$, which represents the access range of j -th dimensional subscript of array A in the tile, is computed as follow.

$$range_j = f_j^{max} - f_j^{min} \quad (1 \leq j \leq m)$$

Here, f_j^{max} and f_j^{min} are the maximum and minimum values taken by subscript expression f_j across the whole tile, respectively.

2. Computing memory range:

mem_range_j represents the number of bytes spanned by the first j dimensions of

array A . It will be computed by $max_address(j,A) - min_address(j,A)$, where $max_address(j,A)$ and $min_address(j,A)$ are the maximum and minimum address values accessed by the first j dimensions. Let dim_size_d denote the d -th dimension size of array A ; $stride_d$, which is the stride of d -th dimension in bytes, is calculated below.

$$stride_1 = [size\ of\ a\ single\ element\ of\ array\ A]$$

$$stride_d = stride_{d-1} \times dim_size_{d-1} \quad (1 \leq d \leq m)$$

Using $stride_d$ and $range_d$, mem_range_j is computed as follow [53].

$$mem_range_j = \sum_{d=1}^j range_d \times stride_d$$

3. Estimating number of distinct lines for one-dimensional array reference:

An exact analysis to compute DL is only performed for array references in which all coefficients are compile-time constants (affine). An upper bound for the number of distinct lines accessed by a one-dimensional array reference [64], or the first dimension of multi-dimensional array, is

$$DL_1 \leq \min \left(\left\lceil \frac{mem_range_1}{L} \right\rceil + 1, \frac{range_1}{g_1} + 1 \right)$$

where g_1 is the greatest common divisor of the coefficients of the enclosing loop indices in f_1 and L is the cache line size in byte. The basic idea is that the DL value is bounded above by the number of cache lines to cover the whole accessed memory range (mem_range_1/L), and the number of distinct accesses ($range_1/g_1$).

4. Estimating number of distinct lines for multi-dimensional array reference:

An upper bound of DL value for the first j dimensions of a multi-dimensional array reference is computed by the following recurrence.

$$DL_j \leq \min \left(\left\lceil \frac{mem_range_j}{L} \right\rceil + 1, \left(\frac{range_j}{g_j} + 1 \right) \times DL_{j-1} \right) \quad (2 \leq j \leq m)$$

DL_m is equivalent to the DL value for the whole dimensions of array A .

In practice we can simplify the above multi-dimensional DL expression as follows. First, find dimension s such that s is the largest dimension to carry spatial data locality on array A , or to satisfy $g_s \times stride_s < L$. The approximated DL value is defined as follow.

$$DL = DL_m \approx \left(\left\lceil \frac{mem_range_s}{L} \right\rceil + 1 \right) \times \prod_{j=s+1}^m \left(\frac{range_j}{g_j} + 1 \right)$$

The relative error is reasonably small when, as is usually the case, all dimensions that are smaller than s also carry spatial data locality on array A . The validity of this approximation is discussed in the past literature [53].

The total number of distinct lines accessed by the whole tile is computed as the summation of individual array references. References on the same array are merged into a single array reference if all their coefficients in the subscript expressions (c_{jk} where $1 \leq j \leq m$, $1 \leq k \leq n$) are same. The difference between minimum and maximum values for constant term of j -th subscript c_{j0} is recorded as *merged_range_j* (e.g., *merged_range* = {1, 2} when $A[i+j][i+k]$ and $A[i+j+1][i+k-2]$ are merged). Using *merged_range_j*, the definition of *range_j* is modified as follow. A more precise discussion is outlined in [64].

$$range_j = \sum_{k=1}^n |c_{jk}| \times (t_k - 1) + merged_range_j \quad (1 \leq j \leq m)$$

The DL definition is also applicable to any level of cache or TLB by selecting its cache line size or page size as L . Unfortunately, the DL model ignores possible replacement of

cache lines and therefore provides conservative upper bounds for the number of cache lines needed.

3.2.2 Parametric Tiling

Although production compilers today have limited tiling capability, there have been significant recent advances in automatic source-to-source code transformation for tiling and several systems for parametric tiling have been developed and made publicly available - TLOG [44], HITLOG [45], and PrimeTile [42]. With such tiled-code generators, it is now possible to generate tiled code for compute-intensive inner kernels (including imperfectly nested loops), that can be tuned to the cache characteristics of the target platform. Thus, just as very effective auto-tuning is currently done for dense linear algebra with ATLAS [55], it becomes feasible to use auto-tuning for user kernels such as stencil-based computations. However, unlike library kernel optimization, exhaustive search of the tile parameter space over several hours to days is generally not attractive for tuning user kernels. This motivates the approach developed in this paper to significantly prune the search space.

3.3 ML: Minimum Working Set Lines

In this section, we introduce a new analytical cost model, ML (Minimum working set Lines), based on the cache capacity required for a tile when intra-tile reuse patterns are taken into account. After defining the ML model, we develop an approach to computing ML for a tile by first constructing a special sub-tile based on analysis of reuse characteristics

and then computing the DL value for that sub-tile. Although we mainly discuss cache capacity in this section, the idea and definition is directly applicable to TLBs by replacing the cache line size by the page size.

3.3.1 Operational Definition of ML

The essential idea behind the ML model is to develop an estimate of the minimal cache capacity needed to execute a tile without incurring any capacity misses, if the pattern of intra-tile data reuse is optimally exploited as described below. Consider a memory access trace of the execution of a single tile, run through an idealized simulation of a fully associative cache. The cache is idealized in that it has unbounded size and an optimal replacement policy where a line in cache is marked for replacement as soon as the last reference to data on that line has been issued (through an oracle that can scan the tile’s entire future references). Before each memory access, the simulator fetches the desired line into the idealized cache if needed. After each memory access, the simulator evicts the cache line if it is the last access (according to the oracle). ML corresponds to the maximum number of lines (high water mark) held in this idealized cache during execution of the entire trace (tile).

3.3.2 Model of Computation

In this work, we focus on the class of affine imperfectly nested loops, where loop bounds and array access expressions are affine functions of the surrounding loop iterators and program constants. For this class of program, it is possible to automatically restructure the code [68, 42] to expose rectangular tiles of parametric size. We assume that a system

such as PrimeTile [42] has already been used to generate parametric rectangularly tiled code; and we focus on the problem of tile size optimization for such codes.

The outermost tiled loop is denoted by $loop_1$, and innermost tiled loop is $loop_n$. Since tiles are rectangular by construction, $loop_i$ ($1 \leq i \leq n$) has the same trip count for any of its executions, it is noted T_i ($1 \leq i \leq n$). The iteration domain of a tile is represented as a tuple $[T_1; T_2; \dots; T_n]$. A tile is surrounded by the loops iterating on all tiles, i.e., tiling loops. For ease of presentation we first assume a single-level of tiling, and then extend our definitions to the multi-level tiling case in Section 3.5.

3.3.3 Distance in Tiled Iteration Space

A specific instance of the loop body is identified by an iteration vector, that is, a coordinate in the iteration space, noted $\vec{p} = (p_1, p_2, \dots, p_n)$. The distance between two iteration vectors \vec{p} and \vec{p}' is expressed as distance vector $\vec{d} = (d_1, d_2, \dots, d_n) = \vec{p}' - \vec{p}$ [69]. Here, \vec{p}' is lexicographically after \vec{p} in the iteration space. For instance, in a tiled loop nest one iteration of the innermost loop $loop_n$ strides over one iteration. We consider that as the shortest scalar distance 1, which is corresponding to distance vector $(0, \dots, 0, 1) = (p_1, \dots, p_{n-1}, p_n + 1) - (p_1, \dots, p_{n-1}, p_n)$. Analogous to tile size tuple, this shortest scalar distance is represented as $[1; 1; \dots; 1]$ and we call this form a sub-tile tuple expression. Also, distance vector $(1, 2, 3) = (p_1 + 1, p_2 + 2, p_3 + 3) - (p_1, p_2, p_3)$ has scalar distance $T_2 T_3 + 2T_3 + 3$, which is represented as $[1; T_2; T_3] + [1; 2; T_3] + [1; 1; 3]$ in sub-tile tuple expression. In general, one iteration of $loop_i$ strides over $size_i$, which is equivalent


```

for (p_1 = [low1 : low1+T_1-1])
  for (p_2 = [low2 : low2+T_2-1])
    for (p_3 = [low3 : low3+T_3-1])
      B[p_1][p_2][p_3] = A[p_1][p_3]
                        + B[p_1 - 2][p_2 - 3][p_3];

```

Figure 3.4 : Sample code for explaining distance in tiled iteration space

to the total number of iteration instances within the loop body of $loop_i$. $size_i$ is defined in both of scalar and sub-tile tuple expression.

$$size_n = 1 = [1; 1; \dots; 1] \quad (i = n)$$

$$size_i = \prod_{j=i+1}^n T_j = [1; \dots; 1; T_{i+1}; \dots, T_n] \quad (i < n)$$

Using size vector $\vec{size} = (size_1, size_2, \dots, size_n)$, we define the *scalar distance* of distance vector $\vec{d} = (d_1, d_2, \dots, d_n)$ and its sub-tile tuple expression as follows.

$$\text{Scalar distance: } \vec{size} \cdot \vec{d} = \sum_{i=1}^n (d_i \times size_i)$$

$$\text{Sub-tile tuple: } \sum_{i=1}^n ([1; \dots; 1; d_i; T_{i+1}; \dots; T_n])$$

3.3.4 Temporal and Spatial Reuse Distance

Temporal and spatial data reuse are expressed with widely used definitions of “reuse distance vectors” (often shortened to “reuse vectors”) [30]. A number of previous efforts introduced methods to compute spatial reuse vectors [30], while temporal reuse vectors are computed from standard dependence analysis using dependence equations. Since a reusable data element, or cache/TLB line, is accessed several times within the loop nest, there can be various definitions of reuse vector, such as *first reuse vector* to represent the distance between closest reusable data accesses and *last reuse vector* to represent the dis-

tance between farthest accesses. In our approach, we calculate all first reuse vectors regarding temporal and spatial reuses within the target loop nest. According to this first reuse distance analysis, we compute the smallest sub-tile within which any reusable data elements and cache/TLB lines are re-accessed at least once. We compute the ML value based on this smallest sub-tile.

Let us consider coordinates \vec{p} and \vec{p}' that access the same array element or cache/TLB line and have data reuse. The dependence equation and past work on computing spatial reuse vector give the following expression to represent all the reuse vectors including first and last reuses.

$$d_{reuse}^{\vec{}} = \vec{p}' - \vec{p} = (p'_1 - p_1, p'_2 - p_2, \dots, p'_n - p_n) = (m_1, m_2, \dots, m_n)$$

Here, m_i is an integer value. $d_{reuse}^{\vec{}}$ must include at least one non-zero m_i element and the first non-zero element must be plus because \vec{p}' comes after \vec{p} in the iteration space. The first reuse vector is equivalent to the lexicographically smallest vector to satisfy the above conditions.

For array A of Figure 3.4, let us consider coordinates $\vec{p} = (p_1, p_2, p_3)$ and $\vec{p}' = (p'_1, p'_2, p'_3)$ that access the same element and have temporal reuse. The dependence equations for input dependence are $p_1 = p'_1$ and $p_3 = p'_3$, and the temporal reuse vector from \vec{p} to \vec{p}' is represented as follow.

$$d_{reuse_1}^{\vec{}} = (p'_1 - p_1, p'_2 - p_2, p'_3 - p_3) = (0, m_2, 0) \quad (p'_2 - p_2 = m_2 \geq 1)$$

Therefore, the first temporal reuse vector for array A is $\vec{d}1 = (0, 1, 0)$ at $m_2 = 1$. Note that the temporal reuse vector $d_{reuse_1}^{\vec{}}$ can be represented as the transitive vector of $\vec{d}1$, and

```

for (p_1 = [low1 : low1+T_1-1])
  for (p_2 = [low2 : low2+T_2-1])
    for (p_3 = [low3 : low3+T_3-1])
      C[p_2 - p_1][p_3 - p_1] = C[p_2 - p_1 - 1][p_3 - p_1 - 1] ...

```

Figure 3.5 : Sample code 2 for explaining reuse distance.

the same array element of array A is accessed in each distance of $\vec{d}1$. The reuse vector for spatial reuse is given in the same manner. Again for array A, the pair of \vec{p} and \vec{p}' access neighboring elements (e.g., A[5][5] and A[5][6]) and have spatial reuse. Analogous to dependence equation, the relationship among \vec{p} and \vec{p}' is represented as $p_1 = p'_1$ and $p_3 + 1 = p'_3$, which are reflected in the following spatial reuse vector from p to p' .

$$d_{reuse_2}^{\vec{}} = (p'_1 - p_1, p'_2 - p_2, p'_3 - p_3) = (0, m_2, 1) \quad (p'_2 - p_2 = m_2 \geq 0)$$

Hence, the first spatial reuse vector for array A is $\vec{d}2 = (0, 0, 1)$ at $m_2 = 0$. For array B, the pair with temporal reuse is (p_1, p_2, p_3) and $(p_1 + 2, p_2 + 3, p_3)$, whose temporal reuse vector is $\vec{d}3 = (2, 3, 0)$. The pair with spatial reuse is (p_1, p_2, p_3) and $(p_1, p_2, p_3 + 1)$, whose spatial reuse vector is $\vec{d}4 = (0, 0, 1)$.

Figure 3.5 is another common case where loop skewing is applied to make the loop nest fully permutable and enable rectangular tiling. Let \vec{p} and \vec{p}' denote coordinates to access same element of array C and have temporal reuse. Since there are two references on array C with different subscripts, flow and anti dependences should be taken into account. Note that input/output dependence which can be handled as transitive dependency of flow and anti dependences does not contribute to the first reuse analysis. Here we start with the case of flow dependence where the dependence direction is from the first reference

$C[p_2 - p_1][p_3 - p_1]$ to the second reference $C[p'_2 - p'_1 - 1][p'_3 - p'_1 - 1]$.

Dependence equations:

$$p'_1 - p_1 = m_1$$

$$p_2 - p_1 = p'_2 - p'_1 - 1 \Rightarrow p'_2 - p_2 = p'_1 - p_1 + 1 = m_1 + 1$$

$$p_3 - p_1 = p'_3 - p'_1 - 1 \Rightarrow p'_3 - p_3 = p'_1 - p_1 + 1 = m_1 + 1$$

Reuse distance vector from \vec{p} to \vec{p}' :

$$d_{reuse_5}^{\vec{}} = (p'_1 - p_1, p'_2 - p_2, p'_3 - p_3) = (m_1, m_1 + 1, m_1 + 1) \quad (p'_1 - p_1 = m_1 \geq 0)$$

Therefore, the first temporal reuse vector is $\vec{d5} = (0, 1, 1)$ at $m_1 = 0$. For the case of anti-dependence where the dependence direction is from $C[p_2 - p_1 - 1][p_3 - p_1 - 1]$ to $C[p'_2 - p'_1][p'_3 - p'_1]$, dependence equations and distance vector are described as follows.

Dependence equations:

$$p'_1 - p_1 = m_1$$

$$p_2 - p_1 - 1 = p'_2 - p'_1 \Rightarrow p'_2 - p_2 = p'_1 - p_1 - 1 = m_1 - 1$$

$$p_3 - p_1 - 1 = p'_3 - p'_1 \Rightarrow p'_3 - p_3 = p'_1 - p_1 - 1 = m_1 - 1$$

Reuse distance vector from \vec{p} to \vec{p}' :

$$d_{reuse_6}^{\vec{}} = (p'_1 - p_1, p'_2 - p_2, p'_3 - p_3) = (m_1, m_1 - 1, m_1 - 1) \quad (p'_1 - p_1 = m_1 \geq 1)$$

We obtain the first temporal reuse vector $\vec{d6} = (1, 0, 0)$ at $m_1 = 1$. Note that $d_{reuse_5}^{\vec{}}$ and $d_{reuse_6}^{\vec{}}$ can be represented as the transitive vector of $\vec{d5}$ and $\vec{d6}$, and as shown in Figure 3.6, the same element of array C is accessed in each distance of $\vec{d5}$ or $\vec{d6}$.

According to Section 3.3.3, we define the *scalar distance* for these reuse distance vectors. For Figure 3.4, the size vector is $\vec{size} = (T_2 T_3, T_3, 1)$. The scalar distance of temporal

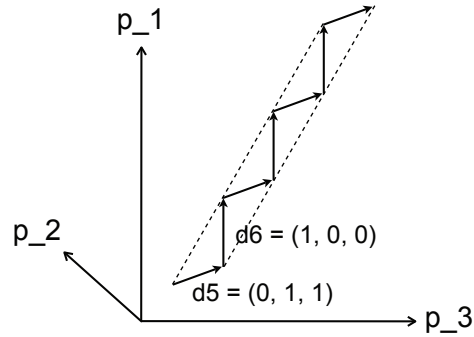


Figure 3.6 : Temporal reuse vectors for sample code 2

reuse vector $\vec{d}1 = (0, 1, 0)$ is calculated by $\vec{size} \cdot \vec{d}1 = T_3$, and the scalar distance of spatial reuse vector $\vec{d}2 = (0, 0, 1)$ is $\vec{size} \cdot \vec{d}2 = 1$. They are also represented as sub-tile tuple expressions $[1; 1; T_3]$ and $[1; 1; 1]$, respectively. Finally, we call the largest scalar distance Maximum Reuse Distance, or MRD. A MRD is defined per array. For example, MRD for array A in the Figure 3.4 is $[1; 1; T_3]$, and MRD for array B is $\vec{size} \cdot \vec{d}3 = (T_2T_3, T_3, 1) \cdot (2, 3, 0) = 2T_2T_3 + 3T_3$, or $[2; T_2; T_3] + [1; 3; T_3]$ in sub-tile tuple expression. Likewise, MRD for array C in Figure 3.5 is $\vec{size} \cdot \vec{d}6 = (T_2T_3, T_3, 1) \cdot (1, 0, 0) = T_2T_3$ or $[1; T_2; T_3]$ in sub-tile expression.

It is still an open question how to approximate non-uniform first reuse vectors such as sparse matrix. As described in section 3.4, ML is used to derive the upper bounds of tile sizes and conservative approximation may make the tile size boundaries smaller than optimal points. Therefore, we ignore non-uniform first reuse so as to estimate tile sizes optimistically, and leave approximation for the non-uniform case to future work. In practice this reuse analysis is individually performed for each merged array reference with

affine form array subscripts (defined in Section 3.2.1) because ML computation is based on the DL expression. In this scenario, array references in a merged group have the same coefficients and first reuse distances are guaranteed to be uniform.

3.3.5 Definition of ML

The sub-tile expression of Maximum Reuse Distance, or MRD, is exactly the target sub-tile for computing ML value. In this section, we show how the ML for each array X is obtained according to its Maximum Reuse Distance. First, a pair of iteration instances (p_1, p_2, \dots, p_n) and $(p_1 + d_1, p_2 + d_2, \dots, p_n + d_n)$ has the following Maximum Reuse Distance for array X.

$$MRD_X = \sum_{i=1}^n ([1; \dots; 1; d_i; T_{i+1}; \dots; T_n])$$

In order to leverage all data locality related to array X, the data at (p_1, p_2, \dots, p_n) must not be removed from the cache memory when $(p_1 + d_1, p_2 + d_2, \dots, p_n + d_n)$ is accessed. It means that the cache memory must keep the all distinct cache lines for array X within the distance of MRD_X . Therefore, ML for array X can be computed as the DL value for the sub-tile tuple expression of MRD_X .

$$ML_X = DL_X(MRD_X) = \sum_{i=1}^n DL(1, \dots, 1, d_i, T_{i+1}, \dots, T_n)$$

As shown in Section 3.3.3, the expression of Maximum Reuse Distance does not include T_1 , and then ML_X is always independent on T_1 . For instance MRD_B and ML_B for array B of Figure 3.4 are computed as follows.

$$MRD_B = [2; T_2; T_3] + [1; 3; T_3]$$

$$ML_B = DL_B(MRD_B) = DL_B(2, T_2, T_3) + DL_B(1, 3, T_3)$$

ML in the tiled loop nest is defined as the summation for all array references.

$$ML = \sum_X (ML_X)$$

In order to leverage all intra-tile data locality within the tile, we should select tile sizes so that ML is smaller or equal to the number of cache lines of the target cache memory, which is usually level-1 cache. Therefore, the tile sizes which satisfy $ML = CacheLines_1$ are the upper boundaries to make full use of intra-tile reuse of level-1 tile.

```

// Tiling loops
for (p2_1 = [0 : M - 1 : T_1])
  for (p2_2 = [0 : N - 1 : T_2])
    for (p2_3 = [0 : K - 1 : T_3])
      // Tiled loops
      for (p_1 = [p2_1 : p2_1 + T_1 - 1])
        for (p_2 = [p2_2 : p2_2 + T_2 - 1])
          for (p_3 = [p2_3 : p2_3 + T_3 - 1])
            C[p_1][p_2] += A[p_1][p_3] * B[p_3][p_2];

```

Figure 3.7 : Matrix multiplication with single-level tiling

3.3.6 Example

Figure 3.7 shows a single-level tiling example for Matrix Multiplication. We assume an element of array has 8 Bytes, and the cache line size of L1/L2 is 64 Bytes (continuous 8 elements are kept in a cache line). DL is calculated as follow.

$$\begin{aligned}
 DL &= DL_C(T_1, T_2, T_3) + DL_A(T_1, T_2, T_3) + DL_B(T_1, T_2, T_3) \\
 &= T_1 \left\lceil \frac{T_2}{8} \right\rceil + T_1 \left\lceil \frac{T_3}{8} \right\rceil + T_3 \left\lceil \frac{T_2}{8} \right\rceil
 \end{aligned}$$

Also, Maximum Reuse Distance for each array is computed from size vector $\vec{size} = (T_2 T_3, T_3, 1)$ as follows.

$$MRD_C = (T_2 T_3, T_3, 1) \cdot (0, 1, 0) = [1; 1; T_3]$$

$$MRD_A = (T_2 T_3, T_3, 1) \cdot (0, 1, 0) = [1; 1; T_3]$$

$$MRD_B = (T_2 T_3, T_3, 1) \cdot (1, 0, 0) = [1; T_2; T_3]$$

Assigning each Maximum Reuse distance to corresponding DL expression, ML for single-level tiling is computed.

$$\begin{aligned} ML &= DL_C(1, 1, T_3) + DL_A(1, 1, T_3) + DL_B(1, T_2, T_3) \\ &= 1 + \left\lceil \frac{T_3}{8} \right\rceil + T_3 \left\lceil \frac{T_2}{8} \right\rceil \end{aligned}$$

3.4 Bounding the Search Space by using DL and ML

This section presents how DL-ML model bounds a tiling search space. As discussed in Section 3.3, ML is used for optimistic cache and TLB capacity constraints for intra-tile data reuse and gives the upper boundaries for estimated tile sizes. In contrast, DL is used for conservative constraints, and gives the lower boundaries. These lower and upper boundaries drastically reduce the search space for single and multi-level tiling. Furthermore, DL, which represents the number of distinct lines within a tile, can be used as a capacity constraint for inter-tile data reuse on higher level of cache/TLB. In this section, we focus on the single-level tiling case which fits within both of level-1 cache and TLB. We extend our approach to multi-level tiling in Section 3.5.

3.4.1 Capacity Constraint for Intra-tile Reuse

Section 3.3.5 shows ML for single-level tiling can be dependent on tile sizes T_2, T_3, \dots, T_n and is independent on T_1 while DL can depend on all tile sizes T_1, T_2, \dots, T_n . CS_1 represents the number of cache lines or TLB entries at level-1 cache or TLB memory. All tile sizes within the lower boundaries due to DL and upper boundaries due to ML satisfy the following constraints.

$$DL(T_1, T_2, \dots, T_n) \geq CS_1$$

$$ML(T_2, T_3, \dots, T_n) \leq CS_1$$

We have two bounded regions according to cache and TLB. In our approach, we consider the union of both regions as candidates for optimal tile sizes, and give higher search priority to the intersection of both regions.

3.4.2 Capacity Constraint for Inter-tile Reuse

Although Section 3.4.1 shows the boundaries to maximize intra-tile data reuse of level-1 tile, the outermost tile size T_1 is actually not bounded above by the ML constraint. As discussed in Section 3.1, this is corresponding to traditional single-level tiling to fit within single-level cache, where the outermost loop is not tiled [46, 70, 48]. However, the outermost tile size affects inter-tile reuse on higher level of cache/TLB, and too large tile size would harm the inter-tile data locality and even the overall performance. Using DL definition, we define an additional capacity constraint so as to preserve inter-tile data reuse on level- k ($k > 1$) cache/TLB as follow.

$$DL(T_1, T_2, \dots, T_n) \leq CS_k$$

This inequality, which ensures the whole distinct lines within the tile can be kept on level- k cache/TLB and guarantees the inter-tile data reuse, gives upper boundary to the outermost tile size T_1 . It is a subject for future work to select the suitable k according to the target system. In the experiments in Section 3.6, we select the highest level of cache/TLB as k .

3.4.3 Empirical Search within Bounded Search Space for Single-level Tiling

Described in previous section, DL-ML capacity constraints for single-level tile consist of the following three conditions.

$$DL(T_1, T_2, \dots, T_n) \geq CS_1 \quad (\text{lower boundary for intra-tile reuse})$$

$$ML(T_2, T_3, \dots, T_n) \leq CS_1 \quad (\text{upper boundary for intra-tile reuse})$$

$$DL(T_1, T_2, \dots, T_n) \leq CS_k \quad (\text{upper boundary for inter-tile reuse})$$

The empirical search finds the optimal tile sizes for T_1, T_2, \dots, T_n , which minimize the objective metrics such as execution time.

Let us calculate the search space of Figure 3.7, which is a single-level tiling example of Matrix Multiplication. We assume the experimental platform has 2-level cache and TLB. The total number of cache lines is 512 for L1 and 49152 for L2 cache, and line size is 64 Byte for both L1 and L2 cache. Program size $N = 3000$ and each array has 8 Bytes per element. Level-1 capacity constraints are

$$DL = T_1 \left\lceil \frac{T_2}{8} \right\rceil + T_1 \left\lceil \frac{T_3}{8} \right\rceil + T_3 \left\lceil \frac{T_2}{8} \right\rceil \geq 512 \quad (\text{L1 cache})$$

$$ML = 1 + \left\lceil \frac{T_3}{8} \right\rceil + T_3 \left\lceil \frac{T_2}{8} \right\rceil \leq 512$$

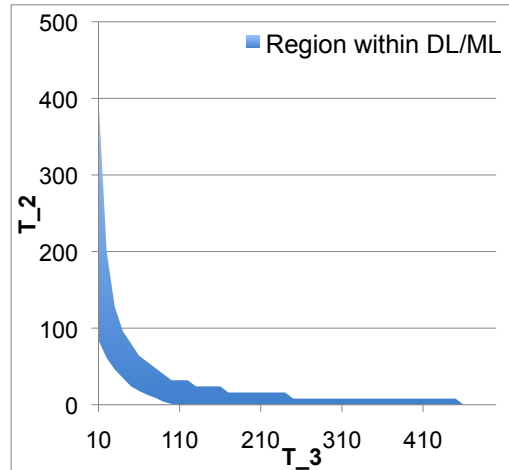


Figure 3.8 : Search space for matrix multiplication for $T_1 = 30$

$$DL = T_1 \left\lceil \frac{T_2}{8} \right\rceil + T_1 \left\lceil \frac{T_3}{8} \right\rceil + T_3 \left\lceil \frac{T_2}{8} \right\rceil \geq 49152 \text{ (L2 cache)}$$

Figure 3.8 shows the bounded search space for (T_2, T_3) when T_1 is 30. These regions bounded by DL/ML constraints are much smaller than the original 2-D search space 3000^2 . The empirical search for level-1 tile finds the optimal tile sizes $T_1 = 100$, $T_2 = 20$ and $T_3 = 120$ within the bounded region.

3.4.4 Compiler Pass for Bounded Search Space

Figure 3.9 shows the compiler framework to implement the DL-ML bounded search space algorithm. This implementation requires standard compiler tools such as dependence vector computation and array index expression extraction, those are readily available in most modern compilers. This is the only program-specific data required to compute the DL-ML equations. Plugging the additional machine-specific information about the different cache level sizes and associated line sizes results in a bounded search space of candidate tile

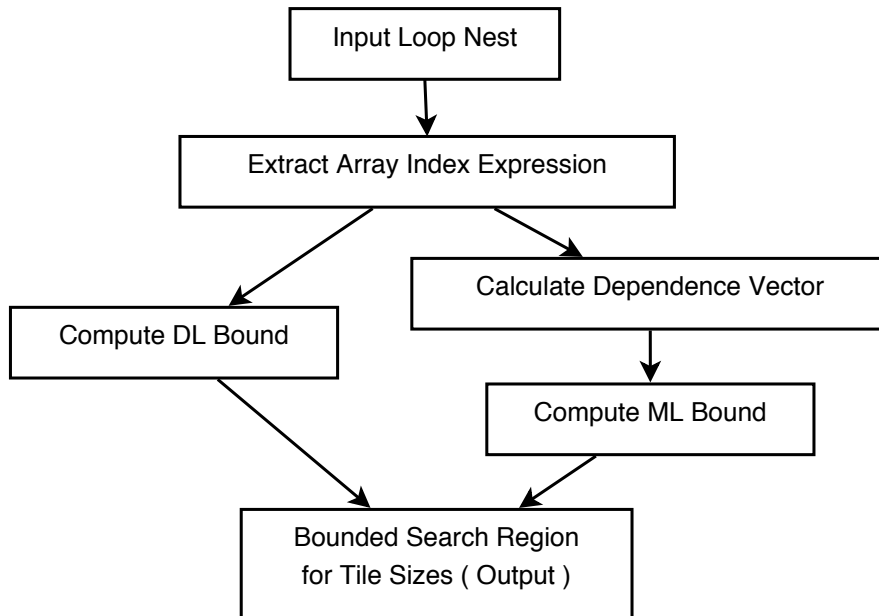


Figure 3.9 : Compiler implementation of DL-ML bounds

sizes, which is drastically smaller than the original set of candidates. Using these bounds, a tile size tuning framework explores only a fraction of points in the original search space, thereby considerably reducing the tuning overhead.

3.5 Extension to Multi-level Tiling

In this section, we extend the DL-ML model to multi-level tiling. We label the innermost loops (for level-1 tiles) as $loop_i^1$ ($1 \leq i \leq n$), the innermost tiling loops, i.e., level-2 tiles by $loop_i^2$ etc. Our approach proceeds by tiling from the innermost level to the outermost, with a level- $(k-1)$ tile being considered as an atomic element at the level- k tile. While it may be intuitive to think of the number of levels of tiling as exactly matching the number

of levels in the hardware memory hierarchy, there are cases when it makes sense to use a smaller number of levels for tiling so the two need not be the same.

3.5.1 Distance in Multi-Level Tiling

A level- k tile consists of the level- k tiled loops $loop_i^k$ ($1 \leq i \leq n$), whose loop index p_i^k iterates over tile size T_i^k with stride T_i^{k-1} . An iteration instance of a level- k tiled loop is identified by the iteration vector $\vec{p}^k = (p_1^k, p_2^k, \dots, p_n^k)$. Figure 3.10 shows an iteration space of a triply nested level- k tiled loops. Each iteration instance corresponds to a level- $(k-1)$ tile represented as a gray cube.

For level- k tiled loops, the distance between two iteration vectors \vec{p}^k and $\vec{p}^{k'}$ is expressed as a distance vector $\vec{d}^k = (d_1^k, d_2^k, \dots, d_n^k) = \vec{p}^{k'} - \vec{p}^k$. Analogous to single-level tiling, the minimum distance vector $(0, \dots, 0, 1)$ corresponds to one stride of the innermost loop $loop_n^k$. Let $size_i^k$ denote the scalar distance for one stride of $loop_i^k$. We define $size_i^k$ as follows:

$$size_n^k = \prod_{j=1}^n T_j^{k-1} = [T_1^{k-1}; T_2^{k-1}; \dots; T_n^{k-1}] \quad (i = n)$$

$$size_i^k = \prod_{j=1}^i T_j^{k-1} \times \prod_{j=i+1}^n T_j^k \\ = [T_1^{k-1}; \dots; T_i^{k-1}; T_{i+1}^k; \dots; T_n^k] \quad (i < n)$$

Using size vector $\vec{size}^k = (size_1^k, size_2^k, \dots, size_n^k)$, we define scalar distance and its sub-tile tuple expression for distance vector $\vec{d}^k = (d_1^k, d_2^k, \dots, d_n^k)$ in the level- k tiled loops.

$$\text{Scalar distance: } \vec{size}^k \cdot \vec{d}^k = \sum_{i=1}^n (d_i^k \times size_i^k)$$

$$\text{Sub-tile: } \sum_{i=1}^n ([T_1^{k-1}; \dots; T_{i-1}^{k-1}; d_i^k T_i^{k-1}; T_{i+1}^k; \dots; T_n^k])$$

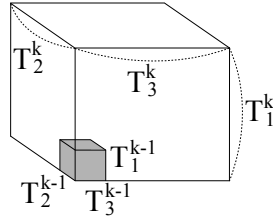


Figure 3.10 : Iteration space for level- k tile

3.5.2 ML for Multi-Level Tiling

Similar to the definition of reuse distance vector for level-1 tiled loops as $\vec{d} = (d_1, d_2, \dots, d_n)$, we define a corresponding level- k reuse distance vector $\vec{d}^k = (d_1^k, d_2^k, \dots, d_n^k)$, where $d_i^k = \lceil d_i / T_i^{k-1} \rceil$. A pair of iteration instances $(p_1^k, p_2^k, \dots, p_n^k)$ and $(p_1^k + d_1^k, p_2^k + d_2^k, \dots, p_n^k + d_n^k)$ has the following Maximum Reuse Distance in the level- k tiled loops.

$$MRD_X^k = \sum_{i=1}^n ([T_1^{k-1}; \dots; T_{i-1}^{k-1}; d_i^k T_i^{k-1}; T_{i+1}^k; \dots; T_n^k])$$

ML for array X within level- k tile is defined as $ML_X^k = DL_X(MRD_X^k)$

Combining the contributions from all arrays, we have:

$$ML^k = \sum_X (ML_X^k)$$

3.5.3 Bounded Search Space for Multi-level Tiling

We consider m -level tiling for an m -level cache hierarchy, where m is the highest cache level. We assume that a level- k tile should fit within both the level- k cache and level- k TLB constraints, where $1 \leq k \leq m$.

Section 3.5.2 shows that ML^k for level- k tiles can be dependent on tile sizes $T_2^k, T_3^k, \dots, T_n^k, T_1^{k-1}, T_2^{k-1}, \dots, T_n^{k-1}$ but not on T_1^k , while DL for level- k tile can depend on $T_1^k, T_2^k, \dots, T_n^k$. Assuming CS_k as the number of level- k cache lines or TLB entries, all tile sizes within the lower boundaries due to DL^k and upper boundaries due to ML^k satisfy the following constraints ($1 \leq k \leq m, T_i^0 = 1$).

$$DL^k(T_1^k, T_2^k, \dots, T_n^k) \geq CS_k$$

$$ML^k(T_2^k, T_3^k, \dots, T_n^k, T_1^{k-1}, T_2^{k-1}, \dots, T_n^{k-1}) \leq CS_k$$

We have two bounded regions according to cache and TLB constraints, and the union of both regions is searched.

Furthermore, we define an additional capacity constraint for a level- k tile so as to preserve inter-tile data reuse with the level- $(k+1)$ cache/TLB.

$$DL^k(T_1^k, T_2^k, \dots, T_n^k) \leq CS_{k+1}$$

These three inequalities correspond to the constraints described in Section 3.4.1 and 3.4.2.

3.5.4 Empirical Search within Bounded Search Space for Multi-level Tiling

In our approach, we first find the optimal tile sizes for level-1 tiles. Using the optimal tile sizes of level-1 tiles, we update the level-2 constraints to bound the search region for level-2 tiles, and so on. We use this decoupled stepwise approach to multi-level tile size optimization because the product search space for multi-level tiling is intractable to fully search even after the significant pruning enabled at each level by use of the DL/ML model. While such a decoupling could result in some good tile combinations to be missed, the

experimental results in Section 3.6.3 demonstrate improvements over single level tiling for most benchmarks.

When the optimal sizes for $T_1^{k-1}, T_2^{k-1}, \dots, T_n^{k-1}$ of level- $(k-1)$ tile have been determined, we update the level- k ML boundary constraint by assigning the optimal sizes to T_i^{k-1} .

$$DL^k(T_1^k, T_2^k, \dots, T_n^k) \geq CS_k$$

$$ML^k(T_2^k, T_3^k, \dots, T_n^k) \leq CS_k$$

$$DL^k(T_1^k, T_2^k, \dots, T_n^k) \leq CS_{k+1}$$

The empirical search finds the optimal tile sizes for $T_1^k, T_2^k, \dots, T_n^k$ within the boundaries recursively. We choose T_i^k to be a multiple of T_i^{k-1} so as to prevent tiles from being fragmented; this also reduces the number of tile size candidates in the search space.

3.6 Experimental Results

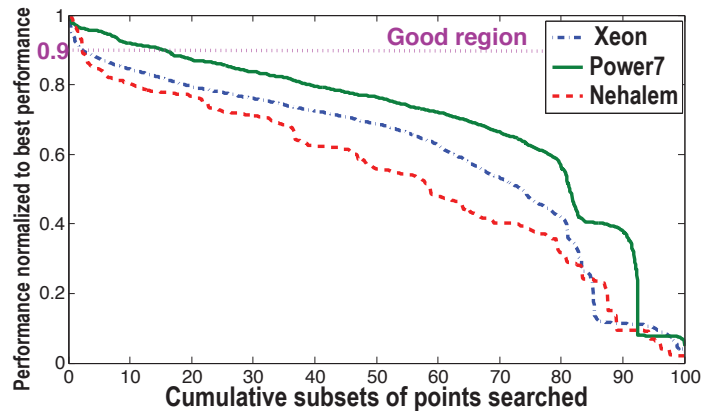
An experimental assessment was performed on three Linux-based systems: an Intel Core i7 920 running at 2.66 GHz with shared L3 cache (labeled *Nehalem*), an IBM Power 7 running at 3.55 GHz (*Power7*), and an Intel Xeon E7330 running at 2.40GHz with shared L2 cache (*Xeon*). Previous work has used published cache capacity data from manufacturers in analytical models for cache performance. However, due to factors such as page table entries, OS processes, etc., the full capacity of higher level caches may not be actually available for use by the application. We report in Table 4.1 the effective capacities for the cache and TLB — the published capacity Spec versus the effective capacity Effective that

was observed using micro-benchmarks for hardware characterization [71]. It may be seen that the effective capacity may be as low as half the documented size, which can affect the DL-ML capacity constraints. In our experiments, we used the effective capacities. The impact of using published capacities instead of effective capacities is studied in Section 3.6.2.

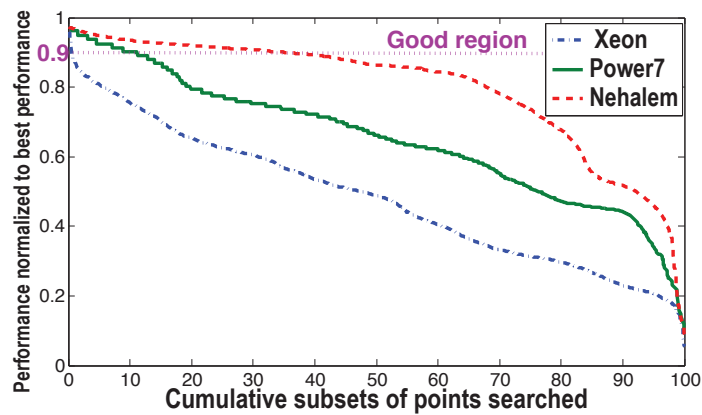
	<i>Nehalem</i>	<i>Power7</i>	<i>Xeon</i>
L1 Spec.	32kB	32kB	32kB
L1 Effective	32kB	32kB	32kB
L2 Spec.	256kB	256kB	3MB
L2 Effective	256kB	256kB	1.5MB
L3 Spec.	8MB	32MB	N/A
L3 Effective	5.2MB	18.4MB	-
Line Size	64B	128B	64B
TLB1 Entries	64	64	16
TLB2 Entries	512	512	256
Page Size	4kB	64kB	4kB

Table 3.1 : Cache characteristics of the architectures considered

We studied five benchmarks using double precision floating point arithmetic. `matmult` is a standard matrix-multiply kernel: $C = A.B$; `dsyrk` is a symmetric rank 1 computation: $C = \alpha.A.A^T + \beta.C$; and `dtrmm` is a triangular in-place matrix multiplication: $B = \alpha.A.B$ (A is triangular). We also considered a representative 9-point two-dimensional stencil computation, `2d-jacobi`, and a 2D Finite Difference Time Domain method, `2d-fdtd`. Parametrically tiled code for each benchmark was generated using the publicly available PrimeTile code generator [42] after any necessary preprocessing such as skewing [68] to ensure that rectangular tiling of the loops was legal. For all tested versions, including the original code, the same compiler optimization flags were used: for *Nehalem* and *Xeon*, we used Intel ICC



(a) matmult-3000x3000



(b) 2d-jacobi-50x4000x4000

Figure 3.11 : Performance distribution for 3.11a matmult-3000x3000 and 3.11b 2d-jacobi-50x4000x4000 on *Nehalem*, *Xeon*, and *Power7*

11.0 with option `-O3`; for *Power7*, we used IBM XLC 10.1 with option `-O3`.

3.6.1 Performance Distribution of Different Tile Sizes

For each benchmark, in the case of single-level tiling, we conducted an extensive set of experiments, for a subset of tile sizes for each loop ranging from 1 to the loop length, in steps of 10 (approximately).

Figure 3.11a plots the data for matmult (size 3000×3000) for the three considered architectures. A point (x,y) in this cumulative plot indicates that $x\%$ of the tile combinations achieved normalized performance greater than or equal to y , where normalization is with respect to the best performing case among all runs and performance is inversely proportional to execution time.

It may be observed that only a small fraction of the tile combinations achieve very good performance — for example, on the *Nehalem*, only 2% of the tile size configurations achieve more than 90% of the maximal performance. Also, there is a very large variation in performance between the best and worst tile size choices, up to a factor of 10. The performance distribution also varies for different targets — for *Power7*, over 20% of the cases provide good performance. Further, we have also observed that the points with good performance are not uniformly distributed in the search space but are clustered in clouds. This highlights the complexity of the search problem when using a blind random search — convergence towards an optimal point may require sampling of a significant fraction of the search space.

Figure 3.11b shows a similar analysis for the 2d-jacobi benchmark. For the target machines, we observe quite a different trend compared to matmult: about 55% of the tile sizes achieve 90% or more of the maximal performance for *Nehalem*, while this ratio significantly decreases for the two other architectures, down to 1% for *Xeon*.

3.6.2 Search Space Reduction by DL-ML Model

To assess the effectiveness of search space pruning by use of the DL-ML model, Figures 3.12-3.16 show the bounded search region superposed with a marking of all tile choices that achieve over 95% of the maximal performance on *Nehalem*, *Power7* and *Xeon*. In each 3-D space, the x, y, and z axes show tile size values for the outer loop, middle loop and inner loop respectively. These tile choices are called “best” points in this section. The surface in each 3-dimensional plot represents the DL-ML upper boundary for single-level tiling, considering intra-tile reuse for level-1 cache and TLB, and inter-tile reuse on the highest level of cache and TLB, as described in Section 3.4. In order to enhance viewability, the figures do not show the lower DL boundaries, since they fall below the best points.

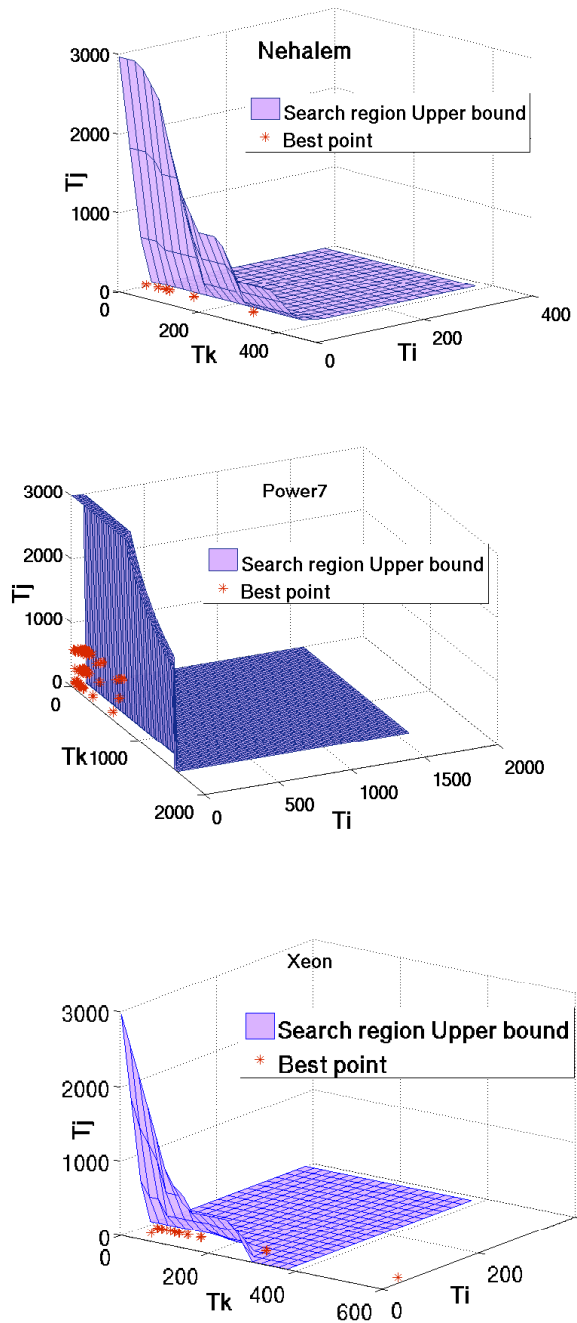


Figure 3.12 : Best tile sizes (achieves 95% or more of the performance with the optimal tile size) for matmult-3000x3000 with $k-i-j$ loop ordering on 3.12a *Nehalem*, 3.12b *Power7* and 3.12c *Xeon*. The x, y and z axes show tile size values for outer loop k , middle loop i and inner loop j respectively.

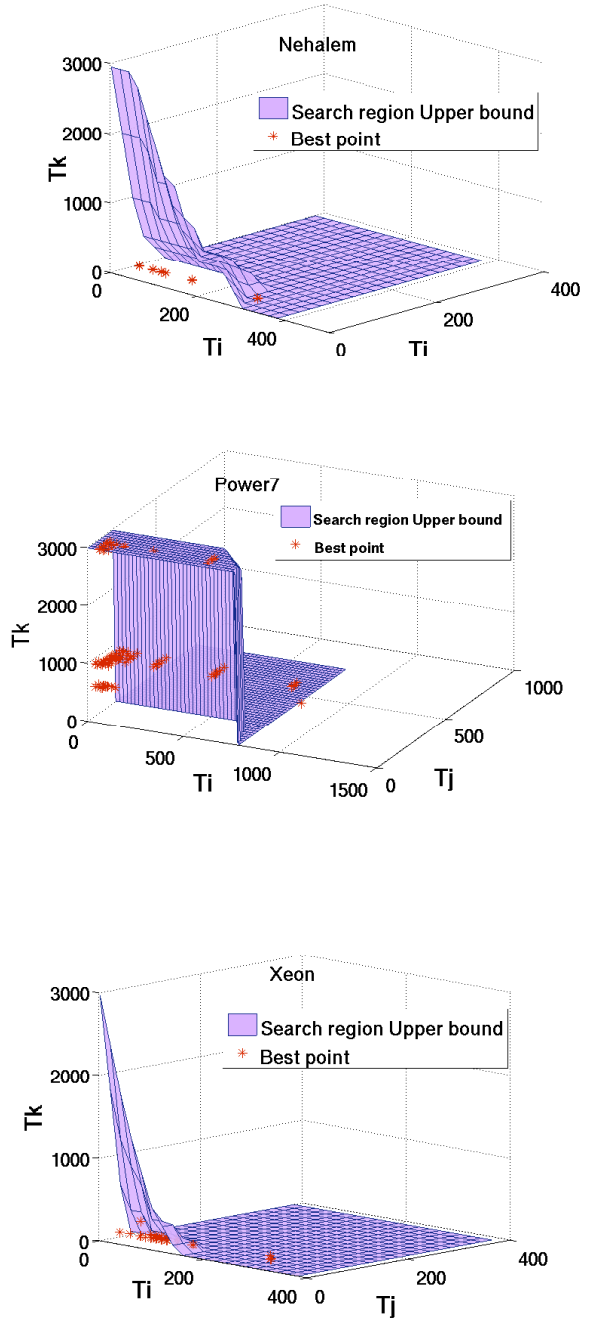


Figure 3.13 : Best tile sizes (achieves 95% or more of the performance with the optimal tile size) for dsyrk-3000x3000 with i - j - k loop ordering on 3.13a *Nehalem*, 3.13b *Power7* and 3.13c *Xeon*. The x, y and z axes show tile size values for outer loop i , middle loop j and inner loop j respectively.

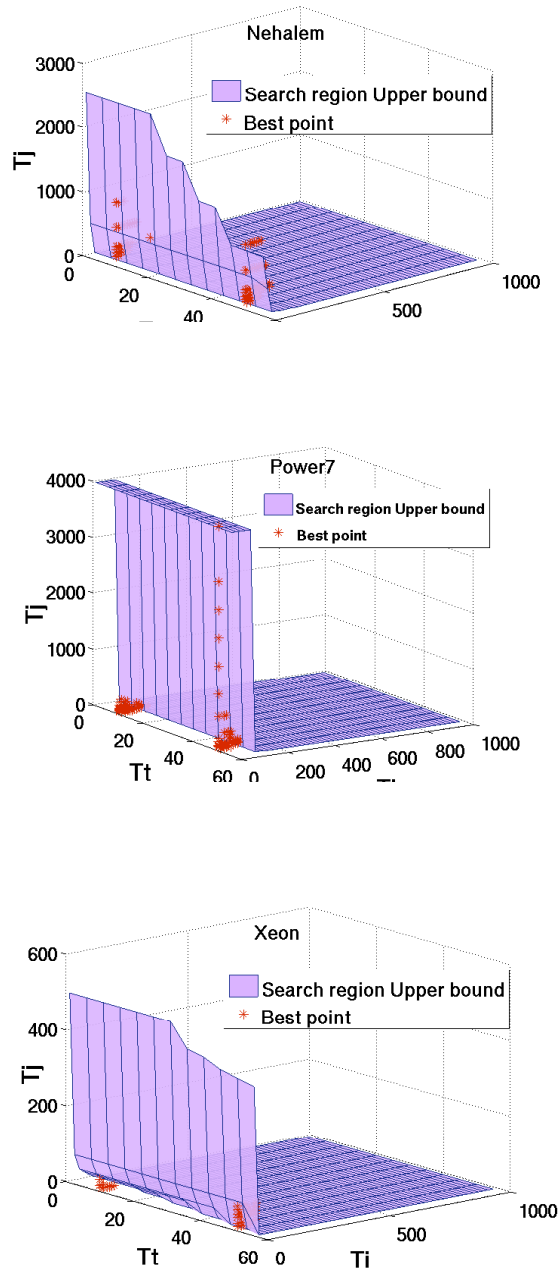


Figure 3.14 : Best tile sizes (achieves 95% or more of the performance with the optimal tile size) for 2d-jacobi-50x4000x4000 with $t-i-j$ loop ordering on 3.14a *Nehalem*, 3.14b *Power7* and 3.14c *Xeon*. The x, y and z axes show tile size values for outer loop t , middle loop i and inner loop j respectively.

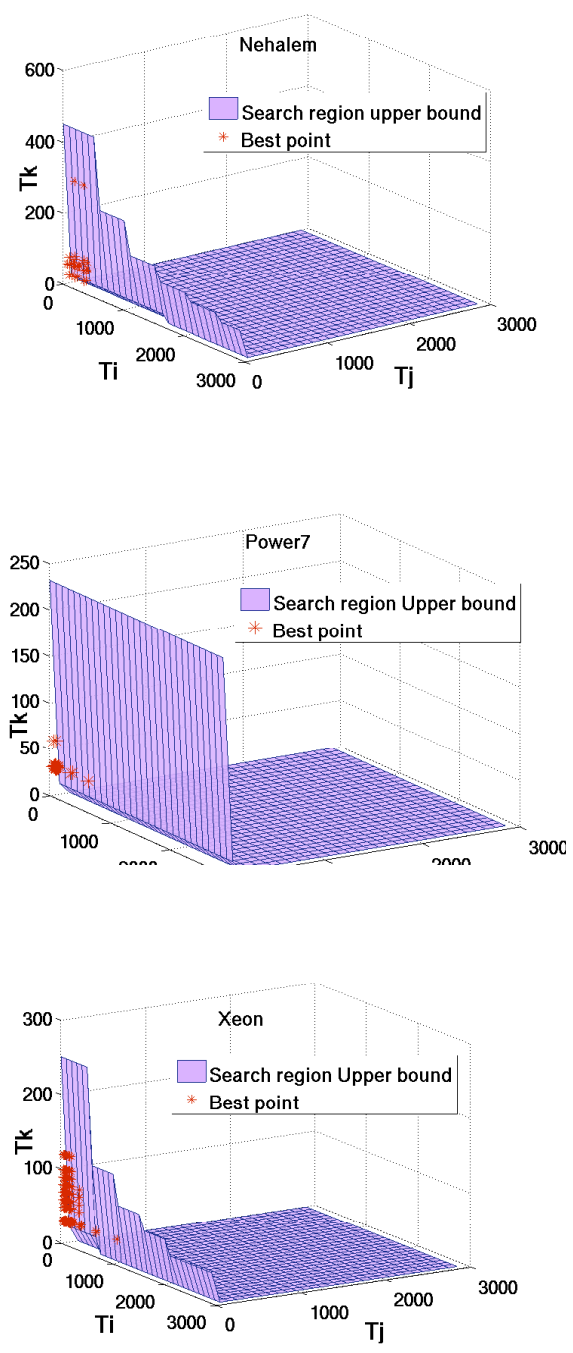


Figure 3.15 : Best tile sizes (achieves 95% or more of the performance with the optimal tile size) for dtrmm-3000x3000 with i - j - k loop ordering on 3.15a *Nehalem*, 3.15b *Power7* and 3.15c *Xeon*. The x, y and z axes show tile size values for outer loop i , middle loop j and inner loop j respectively.

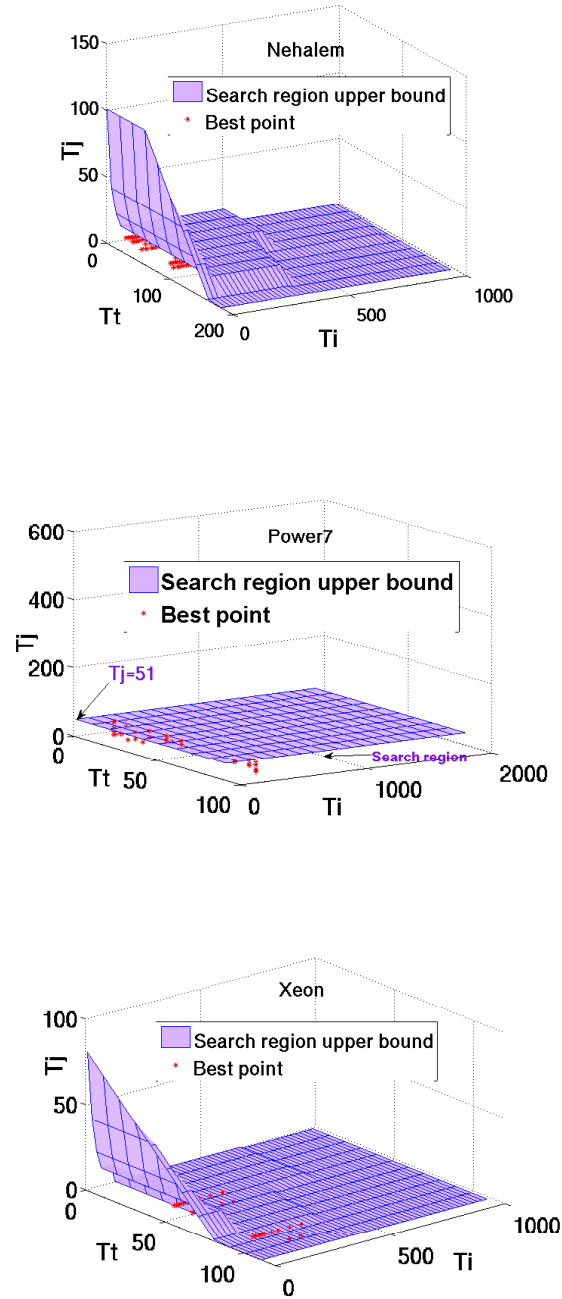


Figure 3.16 : Best tile sizes (achieves 95% or more of the performance with the optimal tile size) for 2d-fdtd-100x2000x2000 with t - i - j loop ordering on 3.16a *Nehalem*, 3.16b *Power7* and 3.16c *Xeon*. The x, y and z axes show tile size values for outer loop t , middle loop i and inner loop j respectively.

For all the plots in Figures 3.12-3.16, we see that although a small number of points lie outside the bounded search space, the vast majority of best points lie inside it. The density of good solutions in the space is thus very much larger than in the non-pruned space. For all the benchmarks, we found that an optimal tile was within the bounded search region. Figures 3.12-3.16 also show that the best tile sizes are relatively smaller on *Nehalem*, and larger on *Power7*. For example, the best points in dsyrk are within the region of ($T_i \leq 400$, $T_j \leq 50$, $T_k \leq 100$) on both *Nehalem* and *Xeon*. However, the best points on *Power7* are distributed much more broadly, up to the maximum size of 3000. This trend pertains to the impact of the level-1 TLB size on each system; the small (4KB) page size on *Nehalem* and *Xeon* causes the best tile sizes to be small, while the large (64KB) page size on *Power7* allows much larger tiles without causing severe TLB misses. These differences are directly reflected in the upper boundary of the DL-ML model, which covers a larger region for *Power7* than the other machines. As discussed in Section 3.4.2, the outermost tile size (x axis) is bounded above only by the inter-tile reuse constraints due to the highest level of cache and TLB. It is obvious that the outermost tile size boundaries also contribute to search space reduction in Figures 3.12-3.16.

Table 3.2 shows the ratio of the space considered in the DL-ML range for the three architectures, compared to the full space of tile sizes. This ratio corresponds to the minimal acceleration factor for an exhaustive or random empirical search compared to using the full space. The factor is much lower for *Power7* due to the larger page size, as explained above. In order to assess the impact of using published versus effective capacities, we repeated

	Problem Size	Xeon	Power 7	Nehalem
matmul	600x600	81.12	1.46	21.90
	3000x3000	8710.81	93.79	1856.49
dsyrk	1000x1000	492.24	2.04	91.62
	3000x3000	11879.67	83.99	1978.26
dtrmm	600x600	41.37	2.31	32.74
	3000x3000	2565.00	1142.23	1238.24
2d-jacobi	50x4000	3102.90	76.43	693.45
2d-fdtd	100x2000x2000	1307.19	45.55	358.74

Table 3.2 : Search space reduction factor across different architectures

our analysis also using the published capacity for the highest level of cache instead of the effective capacity. We found the reduction in search space by use of the ML model was virtually identical with use of published size or effective size (Table 3.2), with one exception: for matmult with 3000×3000 size on *Power7*, the reduction rate decreased from 93.79 to 84.01. This is because the constraint due to the highest level of cache did not affect the search space boundaries for the evaluated benchmarks and platforms except for *Power7/matmult*.

3.6.3 Summary of Experiments

1-Level Tiling

We summarize our experiments for 1-level tiling in Table 3.3. We report, for each benchmark and each architecture, the execution time (in seconds) of the original, untiled code in the Untiled Time column. DL reports the tile sizes and its execution time as obtained by the purely analytical approach using the DL model [64]; Best Square Tile reports the

	Untiled Time	DL		Best Square Tile		Best DL-ML		Impr. by DL-ML	
		Tile Size	Time	Tile size	Time	Tile size	Time	vs. DL	vs. Sq.
matmult-N	33.25	(40, 40, 30)	16.40	(80,80,80)	17.27	(150, 30, 80)	13.48	1.22×	1.28×
matmult-P	25.46	(50, 30, 20)	13.90	(80,80,80)	12.28	(90, 10, 120)	10.60	1.31×	1.15×
matmult-X	153.66	(40, 40, 30)	29.51	(50,50,50)	23.98	(100, 20, 120)	18.35	1.60×	1.31×
dsyrk-N	25.39	(30, 40, 40)	15.47	(80,80,80)	15.54	(30, 30, 90)	12.50	1.23×	1.24×
dsyrk-P	23.32	(40, 30, 30)	15.10	(300,300,300)	10.86	(60, 10, 1000)	9.16	1.64×	1.19×
dsyrk-X	84.89	(30, 40, 40)	26.08	(120,120,120)	25.44	(100, 30, 80)	18.19	1.43×	1.40×
dtrmm-N	142.42	(40, 40, 30)	19.20	(60,60,60)	18.87	(150, 30, 60)	18.20	1.05×	1.04×
dtrmm-P	62.74	(30, 50, 20)	14.60	(60,60,60)	13.06	(600, 30, 32)	11.96	1.22×	1.09×
dtrmm-X	114.70	(40, 40, 30)	28.98	(120,120,120)	29.13	(30, 10, 120)	23.49	1.23×	1.24×
2d-jacobi-N	2.43	(10, 40, 10)	2.60	(50,50,50)	2.24	(10, 8, 150)	2.16	1.20×	1.04×
2d-jacobi-P	2.10	(10, 40, 10)	2.09	(10,50,50)	1.31	(10, 40, 120)	1.19	1.76×	1.10×
2d-jacobi-X	8.75	(10, 40, 10)	2.77	(10,8,8)	2.81	(50, 40, 20)	2.54	1.09×	1.11×
2d-fdtd-N	15.35	(10, 60, 8)	2.41	(50, 8, 8)	2.35	(50, 50, 8)	2.26	1.07×	1.04×
2d-fdtd-P	9.56	(10, 40, 1)	6.90	(50,70,70)	2.11	(40,70,40)	2.09	3.30×	1.01×
2d-fdtd-X	16.42	(10, 60, 8)	4.47	(100,40,40)	4.22	(50,100,8)	4.01	1.11×	1.05×

Table 3.3 : 1-level tiling results (time in seconds, N: Nehalem, P: Power7, X: Xeon)

tile sizes and execution time obtained by an exhaustive empirical search only for square tile sizes; and Best DL-ML is obtained by an exhaustive empirical search in the DL-ML range. The Best DL-ML point was also the globally optimal point in the whole search space for all programs/platforms. We observed that the optimal points represent non-square tile sizes for all cases. For efficient vectorization on all three platforms, the vectorized dimension should correspond to a sufficiently large tile size. Furthermore, the different temporal/spatial data reuse pattern along different dimensions contributes to the unequal sizes of tiles in the different dimensions for the optimal choices.

Empirical search using DL-ML model

This section demonstrates the integration of the analytical bounds with existing search optimization algorithms, the Nelder-Mead Simplex method [72] and the Parallel Rank Or-

dering (PRO) method [73]. To handle boundary constraints due to the DL-ML model, we used the extended version of the PRO algorithm introduced in the Active Harmony framework [58]. The same extension to handle boundaries was employed in our implementation of the Simplex method, and its stopping criteria are based on the work by Luersen [10]. Regarding initial simplex selection for our Simplex search implementation, we used a model-driven approach based on the DL model for square tiling. The square tile size tuple, $T1 = T2 = T3$, which satisfies the DL capacity constraint is selected as one vertex of the initial simplex. Other tree vertices were chosen so as to form a regular triangular pyramid. Note that all the studied kernel loops are triply nested and the simplex always has four vertices. The initial simplex is bounded by the upper and lower tile sizes in addition to the DL-ML bounds.

	Without DL-ML Bounds		With DL-ML Bounds	
	Total [sec]	Best Size / Time [sec]	Total [sec]	Best Size / Time [sec]
matmult-nehalem-simplex	3173.36	(17, 120, 1369) / 13.86	640.98	(36, 56, 64) / 14.71
matmult-nehalem-pro	1294.88	(52, 344, 2270) / 15.64	380.73	(36, 80, 29) / 15.24
matmult-power7-simplex	940.81	(114,1142,858) / 11.4	709.22	(22,82,117) / 11.32
matmult-power7-pro	691.01	(172,1784,2989) / 11.39	442.26	(28,72,126) / 10.58
matmult-xeon-simplex	4268.52	(98,1257,1258) / 21.69	1039.69	(35,56,57) / 19.52
matmult-xeon-pro	2453.03	(97,904,1315) / 21.81	831.73	(31,64,56) / 19.22
2d-jacobi-nehalem-simplex	88.84	(42, 465, 498) / 2.25	26.48	(34,15,64) / 2.32
2d-jacobi-nehalem-pro	51.09	(29, 2001, 2000) / 2.41	50.33	(25,10,627) / 2.2
2d-jacobi-power7-simplex	96.95	(50,37,92) / 1.15	54.94	(50,28,116) / 1.14
2d-jacobi-power7-pro	83.98	(25,8,3495) / 1.61	33.81	(10,53,84) / 1.17
2d-jacobi-xeon-simplex	351.52	(50,40,16) / 2.49	75.49	(50,33,16) / 2.49
2d-jacobi-xeon-pro	248.12	(26,1976,2098) / 8.85	57.34	(10,12,21) / 2.75

Table 3.4 : Empirical search results for 1-level tiling

Table 3.4 shows the total execution time for the whole empirical tuning, the best tile size

	Best DL-ML (2-level tiling)		Perf. Impr. vs. DL-ML (1-level)
	Tile sizes	Time (sec)	
matmult	((3000, 240, 600), (100, 20, 120))	16.28	1.12×
dsyrk	((3000, 600, 240), (100, 30, 80))	17.28	1.05×
dtrmm	((3000, 400, 240), (30, 10, 120))	22.93	1.02×
2d-jacobi	((50, 120, 640), (50, 40, 8))	3.19	1.01×
2d-fdtd	((100, 400, 40), (50, 100, 8))	4.25	0.94×

Table 3.5 : 2-level tiling results on *Xeon* – improvement over 1-level tiling

found by each approach, and its execution time. The DL-ML bounds significantly reduced the total tuning time by a factor of 1.02 to 4.95 on *Nehalem*, 1.33 to 2.48 on *Power7*, and 2.95 to 4.66 on *Xeon*. Furthermore, the Simplex and PRO methods using DL-ML boundary constraints found better tile sizes than the cases without DL-ML bounds, except for the simplex method on *Nehalem*. The tile size search space contains various local optimal points, and these empirical search approaches not using the boundary constraints got stuck at local optimal far from the global optimal point. Note that these search methods can take arbitrary tile sizes in the search space, and hence found slightly better tile sizes in some cases than Table 3.3, which shows the result of scanning the search space with strided access.

Multi-level Tiling

This section presents preliminary experimental results for multi-level tiling. Table 3.5 shows the execution time of the best point within DL and ML boundaries for single and 2-level tiling on *Xeon*. Due to the huge search space for 2-level tiling, we adopted a decoupled approach to model-driven empirical search for tile size optimization. For the first level of tiling, we use the same tile sizes as for single level tiling, based on the lowest level cache/TLB parameters. The tile size candidates for second level tiling were chosen to be multiples of the first level tile sizes, in the space bounded by the DL-ML model based on parameters of the last level cache/TLB. For all benchmarks, 2-level tiling gives better performance than single-level tiling except for 2d-fdtd.

Parallel execution of tiled code

Table 3.6 reports the best tile sizes found by an exhaustive empirical search using DL-ML bounds when the outer-most tiling loop is parallelized with `OpenMP parallel for` directives. It shows the speedup with respect to the untiled sequential execution when running each program with all cores (parallel) and when running with a single core (sequential, same as Table 3.3). Although the performance with parallelization is not always better than sequential, the best tile sizes for parallelized benchmarks also lie in the region bounded by the proposed DL-ML model except for `dtrmm` and `jacobi-2d` on *Xeon*, whose parallel performance is lower than sequential performance. This performance degradation results from inefficient data distribution, which may also cause unexpected effects on tile size selection.

	Optimal Point (parallel)		Optimal Point (sequential)	
	Tile Size	Speedup vs. Untiled Seq.	Tile Size	Speedup vs. Untiled Seq.
matmult-nehalem (8 Threads)	(80, 10, 120)	9.39×	(150, 30, 80)	2.47×
matmult-power7 (32 Threads)	(100, 1, 300)	15.24×	(90, 10, 120)	2.40×
matmult-xeon (8 Threads)	(150, 32, 80)	57.12×	(100, 20, 120)	8.37×
dsyrk-nehalem (8 Threads)	(150, 30, 120)	0.86×	(30, 30, 90)	2.03×
dsyrk-power7 (32 Threads)	(32, 70, 300)	13.79×	(60, 10, 1000)	2.54×
dsyrk-xeon (8 Threads)	(30, 10, 90)	3.64×	(100, 30, 80)	4.66×
dtrmm-nehalem (8 Threads)	(32, 50, 32)	0.93×	(150, 30, 60)	7.83×
dtrmm-power7 (32 Threads)	(10, 30, 100)	1.90×	(600, 30, 32)	5.25×
dtrmm-xeon (8 Threads)	(1, 1, 30)	1.69×	(30, 10, 120)	4.88×
2d-jacobi-nehalem (8 Threads)	(10, 50, 120)	1.77×	(10, 8, 150)	1.13×
2d-jacobi-power7 (32 Threads)	(10, 32, 120)	2.12×	(10, 40, 120)	1.76×
2d-jacobi-xeon (8 Threads)	(10, 40, 600)	3.06×	(50, 40, 20)	3.44×
2d-fdtd-nehalem (8 Threads)	(30, 80, 8)	14.08×	(50, 50, 8)	6.79×
2d-fdtd-power7 (32 Threads)	(10, 80, 8)	15.17×	(40, 70, 40)	4.57×
2d-fdtd-xeon (8 Threads)	(10, 60, 8)	11.99×	(50, 100, 8)	4.09×

Table 3.6 : Parallel 1-level tiling results

3.7 Summary

Our approach to model-driven empirical search for tile size selection is based on a conservative model (DL) that ignores intra-tile cache block replacement, and a new aggressive model (ML) that assumes optimal replacement. In effect, the empirical search can be restricted (pruned) by the two models (DL and ML). Search space reductions ranging from $44\times$ - $11,879\times$ were obtained by using this pruning technique. Our experimental results for single-level tiling on different benchmarks show that almost all of the “best” tile sizes that deliver 95% or more of the optimal performance fall between the ML and DL bounds used in our approach. Furthermore, we demonstrate significant performance improvement for the best tile sizes relative to the tiles sizes obtained by using the analytical DL model

for all five kernels (matmult, syrk, trmm, 2d-jacobi, 2d-fdtd) on the three hardware systems (Nehalem, Xeon, Power7) studied in this work. The experiments for parallel execution show our DL-ML model also works efficiently for tile size selection of parallelized programs. Our results for multi-level memory hierarchy are effective for two-level tiling using DL-ML model on Xeon. Taken together, these experimental results make a convincing case of the effectiveness of our proposed new approach to model-driven empirical search for tile sizes.

Chapter 4

Data Layout Optimization

This dissertation describes a framework and methodology to automatically improve the performance of array-intensive codes running on a variety of computing platforms. As computing platforms increase in diversity, “portable performance” has become one of the most challenging problems for application developers. Achieving good performance on a specific platform often requires coding adjustments to fit a specific set of machine parameters e.g., cache size, cache line size, number of registers, main memory latency, memory bandwidth, etc. Unfortunately, adjustments for one platform can impede performance on other platforms. This dissertation focuses on *data layout optimization*, which has been increasing in importance in recent years. Most programming languages require developers to make array-of-struct (AoS) or struct-of-array (SoA) decisions (or combinations thereof) early in development. For long-lived applications, the following challenge can be encountered repeatedly (and now with increasing frequency): what to do when a new architecture is introduced with a memory subsystem that would benefit from a different data structure layout in the program? With current languages, a near-complete rewrite of an application is usually necessary, since each data access needs to be rewritten. Historically, developers of large codes avoid changing data layouts because it involves changing too many lines of code, the expected benefit of a specific change is difficult to predict, and whatever works

well on one system may hurt on another. Our approach demonstrates how these obstacles can be overcome.

In the next section, we present the data layout framework named TALC. Section 4.2 shows the effect of data layouts across different platform using user-specified layouts. Automated Layout Selection and its experimental results is presented in Section 4.3 and Section 4.4 respectively.

4.1 TALC Data Layout Framework

This section describes our extensions to the TALC Framework [74,75] to support user-specified and automatic data layouts, driven by a Meta file specification. TALC stands for Topologically-Aware Layout in C. TALC is a source-to-source compiler translation tool and accompanying runtime system that dramatically reduces the effort needed to experiment with different data layouts. Our extended version of TALC has been implemented in the latest version of the ROSE [76] compiler infrastructure. In the process of extending TALC, we have re-implemented its entire code, added new functionality for automated layouts and extended layout transformations. The entire source repository can be found online [77].

Figure 4.1 shows the overall framework. TALC can be configured to run in two modes: Automated Layout and User Specified Layout. For both of these modes, a user needs to provide some input to perform data layout transformation. In the Automated Layout mode, the user provides a *field specification*. A field specification file is a simple schema file, which specifies arrays that should be considered for transformation. The field specification file

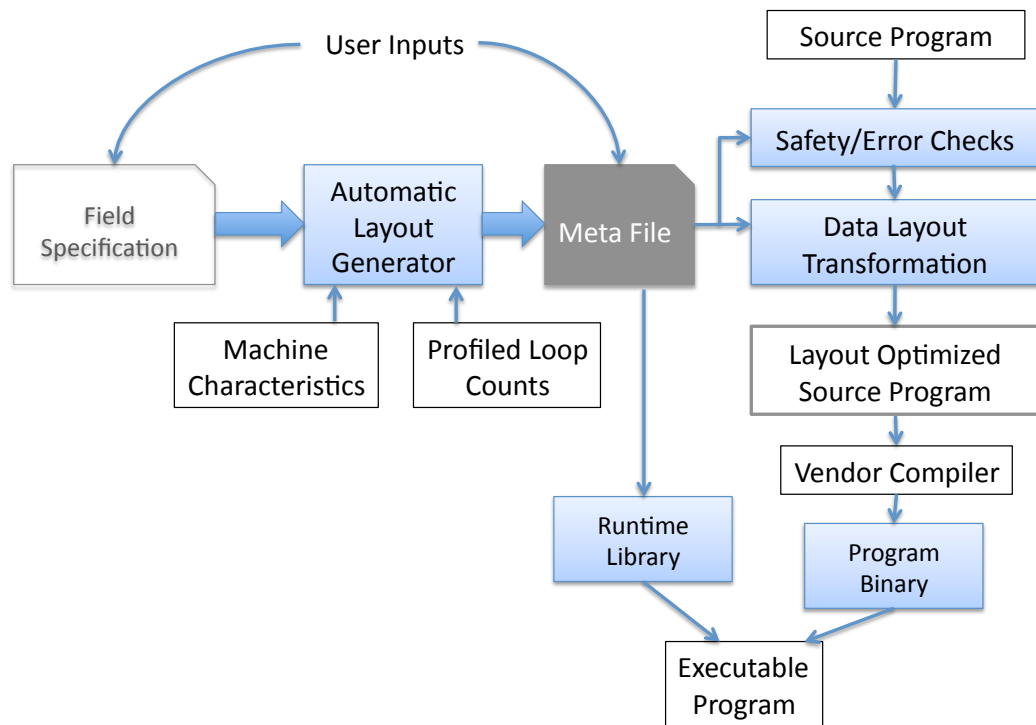


Figure 4.1 : Extended TALC framework

is necessary because it enables our tool to only transform the specified arrays. Figure 4.2 shows a sample field specification file. The View keyword is used internally to parse the data layouts. The field keyword specifies arrays considered for layout transformation. Each field has a type associated with it, specified by the : separator. In this example, d stands for the double data type. Specifying the data type helps with type checking array subscripts during layout transformations. More information on the Automatic Data Layout Selection will be provided in Section 4.3. We focus next on the user specified layout scheme.

The Meta file specifies the data layouts TALC should produce. A Meta file can be generated either automatically or manually. Figure 4.3 contains an example of a Meta

```

View node
{
    Field {x:d}
    Field {y:d}
    Field {z:d}
    ...
}

```

Figure 4.2 : Sample TALC field specification file

```

View node
{
    Field { x:d, y:d, z:d }
    Field { xd:d, yd:d, zd:d }
    Field { xdd:d, ydd:d, zdd:d }
}

```

Figure 4.3 : Sample TALC meta file

file. Unlike the field specification file in Figure 4.2, the Meta file also specifies which fields should be combined into the same array. So, this schema specifies that four arrays of structs are desired. For example, arrays x, y and z will be interleaved in a single array.

Before performing the data layout transformation, we perform safety/error checks for a programmer. These checks not only enable correctness for data layout transformation, but also relieve the programmer from subtle runtime bugs. The safety/error checks perform a pre-pass of the entire program before applying any transformation. A programmer can see the warnings/errors produced by this pre-pass. Following is the list of checks performed by extended TALC framework:

- Type check array fields in Meta file with source program.
- Type check function parameters with array fields in Meta file

- Name check between formal and actual parameters in every function call where parameters match array fields in Meta file

Data Layout transformation is a key component in the TALC framework. The transformation accepts a C/C++ source program and Meta file, produces an equivalent program and changes the data layout of the specified arrays to match the Meta file. The layout transformation matches the names and data type of the arrays before modifying the source code. Array subscripts are automatically rewritten to array and field accesses, as appropriate. The layout transformation also rewrites the memory allocation of the layout transforming arrays to a library call. This call is made at the runtime thereby handling memory allocation gracefully for the entire group in a field. The runtime library ensures memory-aligned allocation for the array grouping. Figure 4.4 shows the key portion of an input file. Figure 4.5 shows a stylized output file (the new array names were inserted for descriptive purposes; the TALC implementation generates synthetic names instead) generated by the layout transformation, based on the Meta file in Figure 4.3.

To ensure that data layout transformations can safely be performed, TALC currently imposes some programming restrictions on the input code:

- All accesses to candidate arrays for data layout transformation must be written as *array[index]*. The alternate form, *index[array]* is prohibited.
- All “aliases” for the same array must use the same name. This is especially important when passing arrays by reference across functions.

```

for (int Node=0; Node<numNode; ++Node) {

    // Calculate new Accelerations for the Node
    xdd[Node] = fx[Node] / Mass[Node];
    ydd[Node] = fy[Node] / Mass[Node];
    zdd[Node] = fz[Node] / Mass[Node];

    // Calculate new Velocity for the Node
    xd[Node] += xdd[Node] * dt ;
    yd[Node] += ydd[Node] * dt ;
    zd[Node] += zdd[Node] * dt ;
    ...
}

```

Figure 4.4 : Sample C input file.

- All arrays in the same field group (as specified in the Meta file) must be of the same length.
- Multidimensional arrays can only be supported if sub-matrices are known to allocated contiguously. The current implementation requires multidimensional subscripts to be transformed to single-dimensional subscripts before data layout transformation. This is a simple transformation for compilers to perform automatically.

We have discussed these limitations with developers of scientific applications at multiple institutions (including LLNL) and have been informed that the assumptions made by the TALC tool are consistent with the assumptions made by the application developers.

```

for (int Node=0; Node<numNode; ++Node) {

    // Calculate new Accelerations for the Node
    Acc[Node].xdd = force[Node].x / Mass[Node];
    Acc[Node].ydd = force[Node].y / Mass[Node];
    Acc[Node].zdd = force[Node].z / Mass[Node];

    // Calculate new Velocity for the Node
    Vel[Node].xd += Acc[Node].xdd * dt ;
    Vel[Node].yd += Acc[Node].ydd * dt ;
    Vel[Node].zd += Acc[Node].zdd * dt ;
    ...
}

```

Figure 4.5 : Stylized TALC output file.

4.2 User Specified Layout Results

We ran a series of tests to evaluate the productivity and performance gains obtained by using TALC to perform layout transformations. In this section we describe the three benchmarks we tested TALC with and the four architectures these tests were performed on. We then describe the manual layouts we tried using TALC and discuss the performance results obtained for the code and architecture combinations that were evaluated.

4.2.1 Test Codes

We ran tests on three codes that are all standard single-node benchmarks containing OpenMP parallelism. A brief overview of each code is described below.

IRSmk

The implicit radiation solver (IRS) [78] is a benchmark used as part of the procurement of the Sequoia system at Lawrence Livermore National Laboratory. It uses a conjugate gradient solver on a 27 point stencil. Included with IRS is a microkernel IRSmk that represents the largest fraction of work in IRS.

SRAD

The SRAD benchmark [79] from the Rodinia suite performs the image processing calculation speckle reducing anisotropic diffusion. The algorithm removes speckles (locally correlated noise) from images without removing image features. We focus on the loop that iterates over the image as it takes almost all of the time in SRAD.

LULESH

The largest application we focus on is the Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) mini-application [80]. LULESH solves the Sedov problem on a 3D hexahedral mesh. Within LULESH there are three categories of arrays that can be combined: node centered, element centered and symmetry. Different array sizes and the fact that they are used in various combinations throughout the loops in LULESH provide a larger search space for data layouts and tension for data layout transformations not found in the smaller benchmarks. The version of LULESH used in this study has undergone a variety of optimizations from the original published code, including aggressive loop fusion

Machine	Architecture Specification
<i>IBM Power7</i>	Quad IBM Power 7 (eight-core 3.55 GHz processor, 32KB L1 D-Cache per core, 256KB L2 Cache, 32MB L3 Cache) Compiler: xlc v11.1 -O3 -qsmp=omp -qthreaded -qhot -qtune=pwr7 -qarch=pwr7
<i>AMD APU</i>	Single AMD A10-5800K APU (quad-core 3.8 GHz processor, 16KB L1 D-Cache per core, 4MB L2 Cache) Compiler: gcc v4.7.2 -O3 -fopenmp
<i>Intel Sandy Bridge</i>	Dual Intel E5-2670 Sandy Bridge CPU (eight-core 2.6 GHz processor, 32KB L1 D-Cache per core, 256KB L2 Cache per core, 20MB L3 Cache) Compiler: icc v12.1.5 -O3 -fast -parallel -openmp
<i>IBM BG/Q</i>	16 IBM PowerPC A2 cores/node, 1.6 GHz processor, 32 MB eDRAM L2 cache Compiler: gcc v.4.4.6 -O3 -fopenmp

Table 4.1 : Architecture and compiler specifications

and elimination of temporary arrays [81].

4.2.2 Experimental Methodology

To show the impact of data layouts on performance we ran experiments using our three test codes on four different platforms. The Linux-based systems are: *IBM Power7*, *AMD APU*, *Intel Sandy Bridge* and *IBM BG/Q*. Table 4.1 summarizes the specifications of these architectures and compiler options used. For the *AMD APU*, we focused on the CPU and ignored the GPU. IRSmk and LULESH were both run in double precision, while SRAD was run in single precision. All were run on varying thread counts on the four platforms. Specifically, we ran IRSmk on a problem based on a 100^3 mesh for 500 iterations. LULESH was run with a problem size = 90 (i.e. 90^3 elements and 91^3 nodes). SRAD was run for 200 iterations on a 4096^2 grid with the x_1 and y_1 speckle values set to 0, the x_2 and y_2 values set to 127 and lambda set to 0.5.

Benchmark	LOC for Base version	LOC (Added+Modified)	
		Min	Max
IRSmk	330	56	272
LULESH	2640	98	477
SRAD	239	11	39

Table 4.2 : Impact of source code lines changes across different layouts compared to base version. LOC denotes Lines of Code.

All of these benchmarks use OpenMP for parallelism. We use the default memory allocation scheme provided in these benchmarks and limit experiments to one socket.

Before showing the specific results, we first comment on the critical impact of TALC on our ability to run these experiments. We calculated the number of source code line changes that were made by the data layout framework across the two benchmarks. Results are tabulated in Table 4.2. Results show the number of source code lines modified or added as compared to the base or original version of source code. These modifications represent the data layout transformation effects on the original code. For IRSmk, the base code is 330 lines and the changes/additions ranged from 56-272 per experiment. For LULESH, the base code is 2640 lines and the changes/additions ranged from 98-477 per experiment. For SRAD, the base code is 239 lines and the changes/additions ranged from 11-39 per experiment. We show the minimum(Min) and maximum(Max) source code changes across the layouts. The upper bound on source line changes are 82% (272 lines) for IRSmk, 18% (477 lines) for LULESH and 16% (39 lines) for SRAD as compared to their base version. These results reiterate the need for having an automated transformation to explore efficient data layouts. By using TALC, we not only were able to automate these changes, but also

Layout	b	dbl	dbc	dbr	dcl	dcc	dcr	dfl	dfc	dfr	cbl	cbc	cbr	ccl	ccc	ccr	cfi	cfc	cfr	ubl	ubc	ubr	ucl	ucc	ucr	ufl	ufc	ufr								
1																																				
2	Group 1			Group 2			Group 3			Group 4			Group 5			Group 6			Group 7			Group 8			Group 9											
3	Group 1																																			
4	Group 1																																			
5	Group 1									Group 2									Group 3																	
6	Group 1														Group 2																					
7	Group 1														Group 2																					
8	Group 1														Group 2																					
9	Group 1									Group 2									Group 3									Group 4								

Figure 4.6 : IRSmk layouts selected for discussion.

eliminate the possibility of subtle bugs when these changes are performed manually.

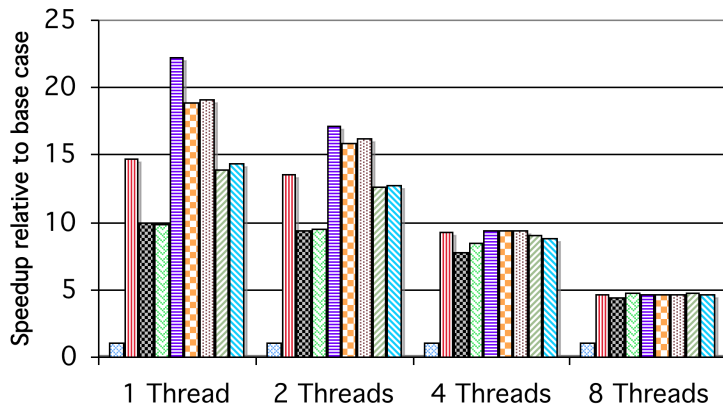
4.2.3 Experimental Results

For each benchmark, we conducted extensive experiments across different layouts on four architectures. Although we have experimented with a larger set of layouts, we limit the number of layouts presented here to the most interesting ones.

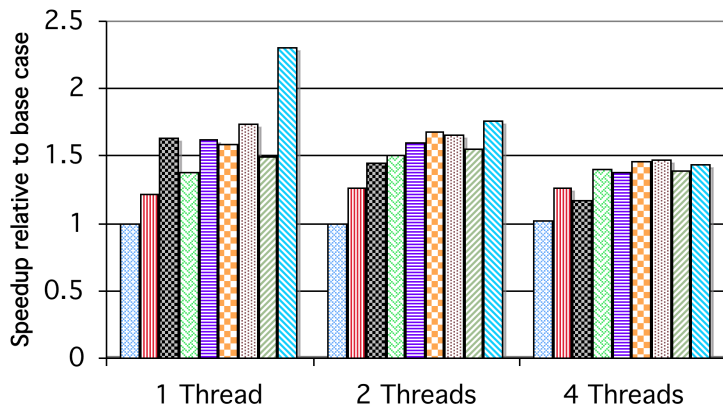
IRSmk

Figure 4.6 shows the nine IRSmk layouts for which we present results. Each row "block" represents a set of arrays (identified by column) that have been interleaved. So, for example, layout 2 consists of nine arrays of structs containing three fields each. The names in each block are added for clarity of presentation. Figures 4.7 and 4.8 shows the results obtained by running IRSmk with different thread counts on all nine layouts on each of the four platforms. For each test case, we report the speedup (or slowdown, for *values* < 1) of each layout against the "base case" which is the original code, running with an equiv-

Layout 1 Layout 2 Layout 3 Layout 4 Layout 5 Layout 6 Layout 7 Layout 8 Layout 9



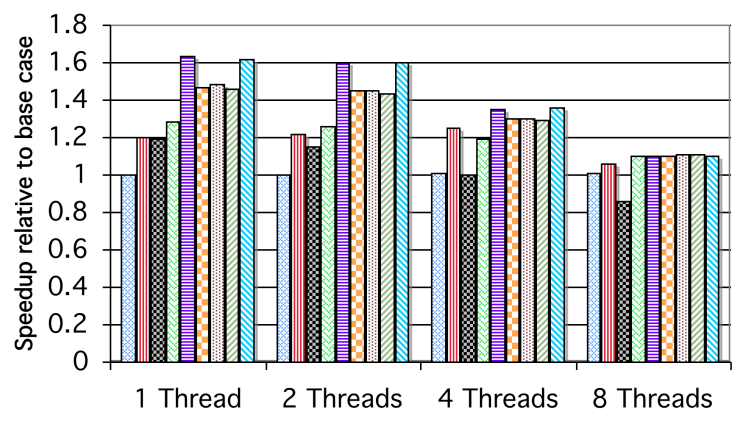
(a) IBM Power7



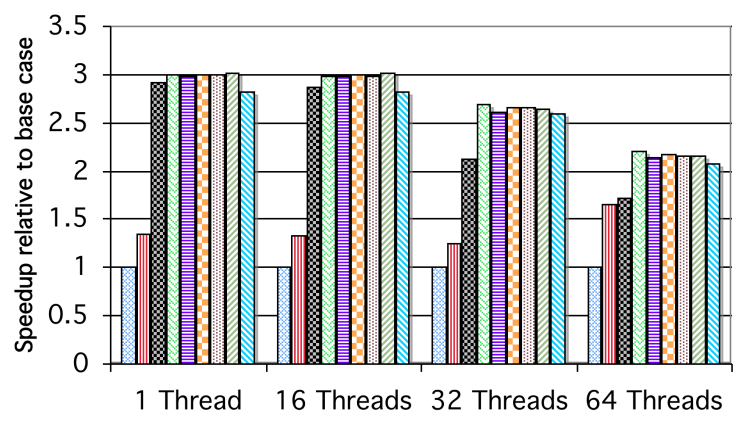
(b) AMD APU

Figure 4.7 : IRSmk performance results on IBM Power 7 and AMD APU platforms with varying threads.

Layout 1 Layout 2 Layout 3 Layout 4 Layout 5 Layout 6 Layout 7 Layout 8 Layout 9



(a) Intel Sandy Bridge



(b) IBM BG/Q

Figure 4.8 : IRSmk performance results on Intel Sandy Bridge and IBM BG/Q platforms with varying threads.

```

#pragma omp parallel for private(jj,ii,i)
for (kk = kmin; kk < kmax; kk++) {
  for (jj = jmin; jj < jmax; jj++) {
    for (ii = imin; ii < imax; ii++) {
      i = ii + jj * jp + kk * kp;
      b[i] = dbl[i] * xdbl[i] + dbc[i] * xdbc[i] + dbr[i] * xdbr[i] +
        dcl[i] * xdcl[i] + dcc[i] * xdcc[i] + dcr[i] * xdcr[i] +
        dfl[i] * xdfl[i] + dfc[i] * xdfc[i] + dfr[i] * xdfr[i] +
        cbl[i] * xcbl[i] + cbc[i] * xcbc[i] + cbr[i] * xcbr[i] +
        ccl[i] * xccl[i] + ccc[i] * xccc[i] + ccr[i] * xccr[i] +
        cfl[i] * xcfl[i] + cfc[i] * xcfc[i] + cfr[i] * xcfr[i] +
        ubl[i] * xubl[i] + ubc[i] * xubc[i] + ubr[i] * xubr[i] +
        ucl[i] * xucl[i] + ucc[i] * xucc[i] + ucr[i] * xucr[i] +
        ufl[i] * xufl[i] + ufc[i] * xufc[i] + ufr[i] * xufr[i] ;
    }
  }
}

```

Figure 4.9 : IRSmk source code

alent number of threads. In some cases, for example IBM BG/Q for 2, 4 and 8 threads, we omitted thread counts because their results were similar to adjacent thread counts of 1 and 16. From these figures, we clearly see speedup from $1.10\times$ - $22.23\times$. Improvements are more for lower thread count as compared to higher thread count since there is more effective cache capacity at higher thread counts for the same problem size. This trend was observed across all benchmarks.

To understand how layouts impact the performance of IRSmk, one first needs to understand that this kernel is memory bound. Each iteration of the innermost loop in Figure 4.9 reads one unique double from 27 arrays, writes a single value to the b array and may read data from other parts of x from memory if it is not in cache already. Array references

starting with 'x' alias to the same array with different offsets. Since each array is about 9MB, other than x , there is no chance of any array staying within cache between iterations. Therefore, performance should be limited by memory bandwidth. However, we see that except for Sandy Bridge significant speedups occur at all thread counts due to data layouts.

To better understand what is happening, we examined the total bandwidth requirement for moving all 29 arrays either from or to memory. For the 500 iterations, this is about 119 GB of data motion. For now we ignore the fact that x might be moved multiple times from memory, assume good caching to obtain an upper bound on performance, and determine the bandwidth limited runtime of each system using the Stream Triad numbers [82]. For Sandy Bridge, which has about 40 GB/s stream bandwidth this is just under 3.0 seconds. For BG/Q with about 28 GB/s of bandwidth, this is 4.3 seconds. For the AMD APU with about 15 GB/s of bandwidth, this is just under 8.0 seconds. For the Power 7, there is about 13 GB/s of bandwidth, implying a lower bound of 9.2 seconds. Therefore, we have a best case runtime for each machine.

The results of the best layout for IRS on all machines show performance of at least 70% of optimal and over 95% on Sandy Bridge. For Sandy Bridge, the execution time for the best layout is 3.05 seconds, for the AMD APU it is 10.04 seconds, for BG/Q it is 5.2 seconds and for the Power 7 it is 12.52 seconds. BG/Q performs slightly worse than other architectures due to in-order cores not hiding as much latency as the other processors, while the AMD APU could be hurt by less data in the x array staying in its smaller cache. Finally, all the processors might be limited in their handling of the unequal amount of read

and write data in IRSmk.

While the best case scenarios for each processor are similar in their efficiency, the efficiency of the base case is significantly different. On the Sandy Bridge, data layouts only sped up the computation by $1.11\times$ as the base case was already running at over 85% of peak memory bandwidth. On the other processors, performance is significantly worse for the base case. From looking at the hardware specifications the largest difference in this regard is the number of hardware prefetch streams each processor can keep active at once. The Sandy Bridge can handle 32 per core [83], while BG/Q can handle 16 per core [84], the Power7 can handle 8 per core to the L2 and 24 per socket to the L3 [23] and the AMD APU can handle 12 per core [85]. Therefore, the Sandy Bridge processor can handle all the arrays in the computation at once. However on other processors fusing arrays decreases the number of streams coming from memory, resulting in fewer data elements read in a latency-bound manner. This is especially important for in-order cores such as BG/Q. Also the Power 7 benefits significantly from fewer arrays. Other layouts that are not shown in these figures indicate a trend of steadily improving performance as more arrays are grouped. A single array of struct(AoS) is generally not the best case for an application since access patterns in segments of the code may not use all the arrays. Also when a calculation is latency bound, such as the base case, doubling the thread count halves runtime, by doubling the number of latency bound reads occurring concurrently. However, the memory bound layouts, that do not use all the prefetchers have their runtime barely decrease, such as layout 5, from 2 to 8 threads.

An important observation is that improvements from data layouts are more significant at lower core counts. This implies two conclusions. First compute-bound codes also benefit from data layout transformations. In the case of Sandy Bridge where there are enough stream prefetchers for the base code and enough bandwidth to feed a few, but not, all, cores merging arrays reduces the number of registers used as pointers by the compiler resulting in fewer instructions and possibly fewer pipeline stalls. Another benefit is that the number of elements accessed in each loop from an array can be matched to cache boundaries, such as layout 9. The second observation is that for processors with an under provisioning of prefetchers when fewer cores are used the computation becomes latency-bound. With fewer cores to issue memory requests, the memory bus becomes idle for a larger percentage of the time. Therefore, bandwidth is used less efficiently, allowing for larger speedups when the core uses it more effectively.

A final observation is that not merging read only arrays in a loop with arrays that are written increases the performance significantly. Modern architectures, such as AMD APU, often implement a write buffer to combine multiple writes to the same cache line to reduce the amount of data sent to main memory. This optimization is known as Write-Combining [86]. For IRSmk, when b was not merged with other groups, performance was better in all cases (except some single threaded examples) as compared to combining it with other arrays. The performance difference between layouts 3 and 4 illustrates this phenomenon, as an example.

Layout	I	iN	iS	jW	jE	C	J	dN	dS	dW	dE
1											
2											
3											
4											
5											

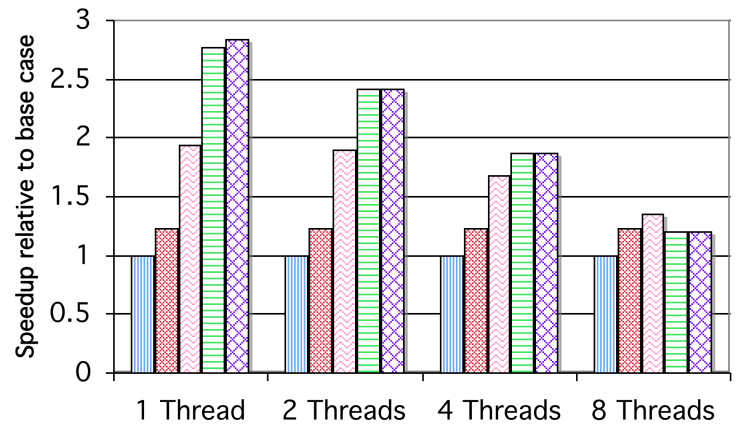
Figure 4.10 : SRAD layouts selected for discussion.

SRAD

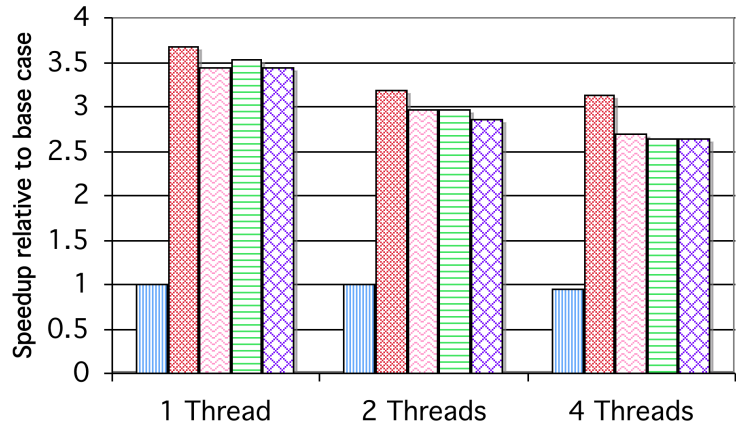
Figures 4.11 and 4.12 show SRAD results for running the five layouts presented in Figure 4.10 on our four test platforms. SRAD contains many of the same trends and results as IRSmk, but adds some new features and complexity. SRAD contains multiple loops so there are cases where two arrays are used together in one loop and only one array is used in another loop. Examples of this are the *IN*, *IS* loop pair and the *JW*, *JE* set of loops. In addition, some of the arrays such as *J* are accessed in multiple places in the same loop in SRAD, but can be combined with other arrays, such as *DN*, which are only accessed in one place. Finally, since SRAD is run on more compute intense problems vectorization can increase its performance significantly. Our results show how some of these tensions impact performance.

For example, if we compare layout 5 and layout 4, layout 5 combines more array references across different loops and outperforms layout 4, yet layout 5 is not the best layout as the array references combined appear to benefit limited number of loops. An example

Layout 1 Layout 2 Layout 3 Layout 4 Layout 5



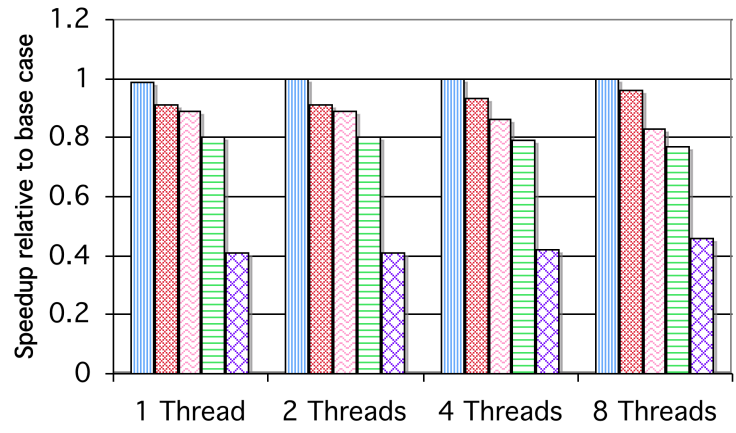
(a) IBM Power7



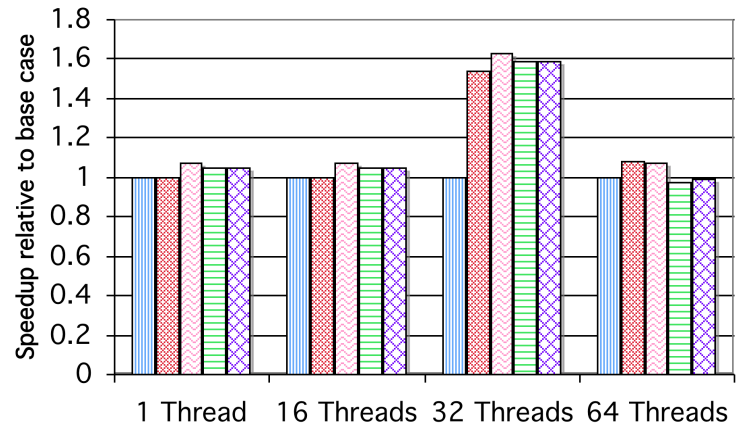
(b) AMD APU

Figure 4.11 : SRAD performance results on IBM Power 7 and AMD APU platforms with varying threads.

Layout 1 Layout 2 Layout 3 Layout 4 Layout 5



(a) Intel Sandy Bridge



(b) IBM BG/Q

Figure 4.12 : SRAD performance results on Intel Sandy Bridge and IBM BG/Q platforms with varying threads.

of this effect is seen on BG/Q platform, where combining the two arrays when using 64 threads increases performance. There are two possible explanations. First by combining these pairs the number of streams per hardware thread is reduced to 4, which is the most that can be fetched on this machine. Second, if the code is not memory bound then the extra data motion caused by the combination does not negatively impact performance when only one element is used. Since the only other machine to see a performance gain is the Power7 at a low thread count when memory bandwidth is plentiful the first explanation is likely the reason and the gains in one loops are overcoming the negative impacts in the other.

When combining arrays accessed in multiple places with arrays only accessed in one place, the results are mixed. In some cases the combination helps performance, while in others it hurts performance. Layouts two and three allow a simple comparison of this. What is seen is that on thread counts below 64 on BG/Q and on the Power 7 that combining the arrays is helpful. However, on other machines and large BG/Q, thread counts it is better to keep the arrays separate. What is happening is that in some cases the data are staying in cache long enough to be reused and in other cases poor caching along with data transformations is causing extra data motion. Note the caches with the worst performance are all 512 KB or less per thread and with perfect caching the loop with the most data would need to hold at least 400 KB of data to prevent extra reads from memory.

Finally, on the Sandy Bridge chip with the Intel compiler SRAD gets a significant performance boost from vector instructions. However, when data layout transformations are performed the compiler no longer vectorizes any instructions due to the use of pointers

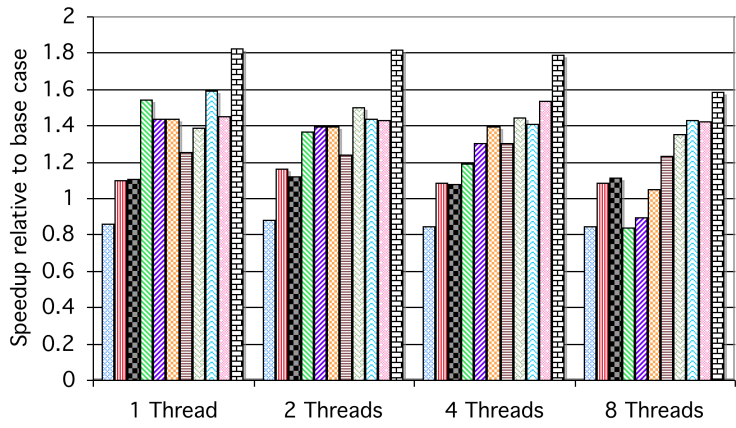
Layout ID	Nodal Arrays													Symmetry			Element Arrays																			
	x	y	z	xd	yd	zd	xdd	ydd	zdd	fx	fy	fz	nodalMass	symmX	symmY	symmZ	ixim	ixip	letam	letap	izetam	izetap	ql	qq	p	q	elemBC	e	ss	elemMass	delv	volo	v	vdov	arealg	
1																																				
2	Group 1				Group 2				Group 3				Group 4				Group 5																			
3	Group 1				Group 2				Group 3				Group 4				Group 5			Group 6																
4	Group 1													Group 2			Group 3																			
5	Group 1													Group 2			Group 3				Group 4															
6	Group 1				Group 2				Group 3				Group 4				Group 5			Group 6																
7	Group 1				Group 2				Group 3				Group 4				Group 5			Group 6			Grp 7		Grp 8											
8	Group 1				Group 2				Group 3				Group 4				Group 5			Group 6			Grp 7		Grp 8		Grp 9		Grp 10		Grp 11		Grp 12			
9	Group 1				Group 2				Group 3				Group 4				Group 5			Group 6			Grp 7		Grp 8		Grp 9		Grp 10		Grp 11		Grp 12			
10	Group 1				Group 2				Group 3				Group 4				Group 5			Group 6			Grp 7		Grp 8		Group 9		Grp 10		Grp 11		Grp 12			
11	Group 1				Group 2				Group 3					Group 4			Group 5						Group 6			Grp 7										

Figure 4.13 : LULESH layouts selected for discussion.

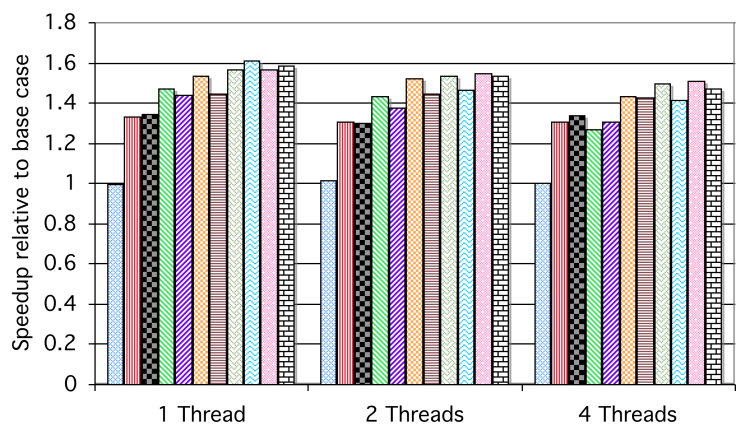
to the structures. The result is a performance hit from vectorization that is greater than the gain from data layout transformations. To confirm this we ran the base version of SRAD with compiler vectorization turned off and data layout transformations resulting in a 1.66x to 1.84x speedup from data layout transformations.

Overall, though performance gains on SRAD ranged from the minor 1.07x on most BG/Q thread counts to 3.68x on a single thread of an AMD APU. With three different layouts being best depending on architecture there is no clear good cross platform layout for SRAD. It is important to note the impact of prefetching on SRAD as well. For example on BG/Q performance gains are slight except for 32 threads, where all the layouts tried use 8 or fewer prefetchers per threads while the initial code uses more.

Layout 1 Layout 2 Layout 3 Layout 4 Layout 5 Layout 6 Layout 7 Layout 8 Layout 9 Layout 10 Layout 11



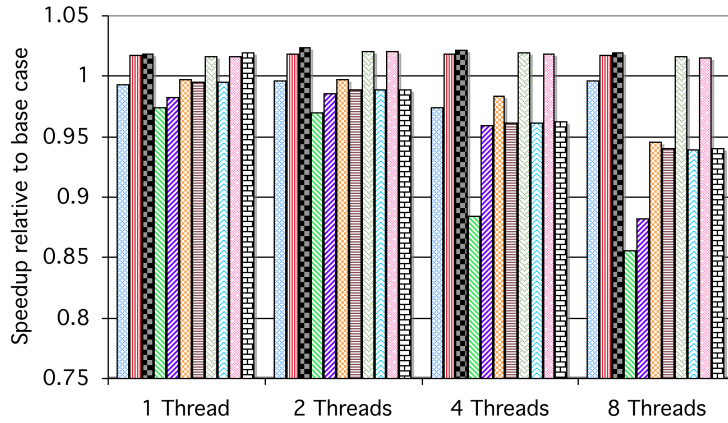
(a) IBM Power7



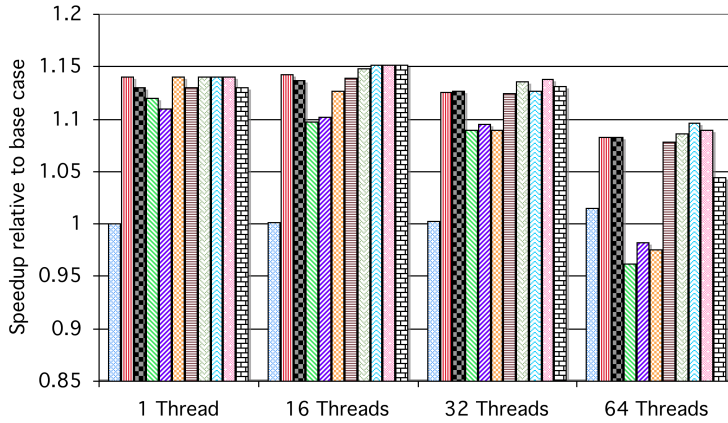
(b) AMD APU

Figure 4.14 : LULESH performance results on IBM Power 7 and AMD APU platforms with varying threads.

Layout 1 Layout 2 Layout 3 Layout 4 Layout 5 Layout 6 Layout 7 Layout 8 Layout 9 Layout 10 Layout 11



(a) Intel Sandy Bridge[†]



(b) IBM BG/Q[†]

Figure 4.15 : LULESH performance results on Intel Sandy Bridge and IBM BG/Q platforms with varying threads.

[†] To show performance variation, we start scales from non-zero values.

LULESH

Figures 4.14 and 4.15 show LULESH results for running the eleven layouts presented in Figure 4.13 on our four test platforms. As with IRSmk the number of threads was varied for each platform and we only show selected interesting thread counts. For each test case, we report the speedup (or slowdown) of each layout against the “base case” which is the original code, running with an equivalent number of threads.

Data layout transformations on LULESH were less profitable overall than for IRSmk. This is not surprising since LULESH is a larger application than IRSmk, and some arrays in LULESH are used together in certain places and not together in others. Therefore, combining them together will help and hurt performance simultaneously. For example, layout 4 combines all four triples of x, y, z values together. Many of these triples are used together in many functions, but not all. However, most of the time layout 6 which leaves the triples separate is faster. A notable exception can be seen on Power7 for a single thread, which has the most cache, but the least bandwidth. It also suffers the most from not getting good prefetching as shown by the IRSmk results.

The most interesting result from LULESH is that in most cases it seems the code not the hardware is dictating the best data layout. On the AMD APU, Intel Sandy Bridge and BG/Q the list of the best layouts always includes 8 and 10 and usually, includes 2 and 3. However, the Power7 is an outlier with its best layout being 11 for all thread counts by a significant margin for the reasons explained above.

For LULESH, as with IRSmk and SRAD, data layouts impacted the Sandy Bridge

system the least with the largest speedup seen being only $1.02\times$. There are a few likely reasons for this. First, as with IRSmk, the Sandy Bridge architecture should be able to prefetch many streams at once. Also, in the case of bundling indirect accesses, the large reorder window of the Sandy Bridge might hide memory latency better than the other chips. Finally, the Intel compiler used on this platform was the best at generating SIMD instructions for some of the compute bound loops of LULESH. Some of the data transformations result in the compiler no longer generating SIMD instructions and, therefore, while data layouts save on data motion in memory-bound portions of the code they can sometimes hurt performance in the compute bound sections.

4.3 Automatic Data Layout Selection

In this section, we describe the automatic data layout selection algorithm. The algorithm takes in a user-written field specification file and uses a greedy algorithm to automatically construct a data layout based on the architecture and input program. We first describe the use graph and cache-use parameter used by the algorithm, and then the algorithm itself.

4.3.1 Use Graph

The automated analysis begins by creating a mapping of all arrays used within each loop of the source program. We are only interested in determining which arrays appear inside each loop, not the exact location or order of use. So, our use graph is a mapping from each array name to a function name along with the loop for the reference. In the case

of nested loops, each array points to the inner-most loop in which the reference occurs. Multiple references to the same array in a loop are not distinguished; however, we do keep track of the types of accesses that occur: read, write, read/write. If a reference does not appear in a loop, we only use reference's enclosing function name.

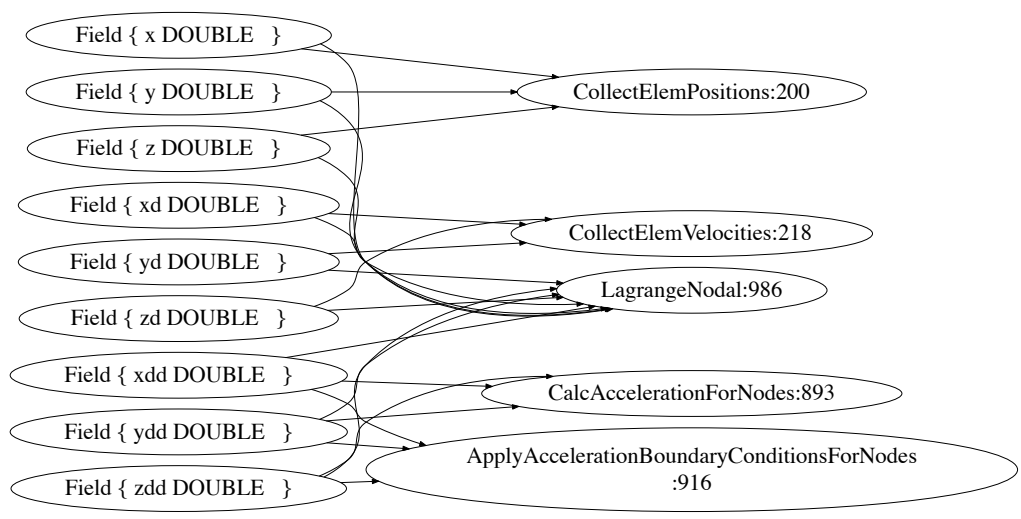


Figure 4.16 : Sample use graph

The use graph is a bipartite graph $G=(U,V,E)$ where U is the set of arrays in the field specification, V is a set of uses of array references (a static array reference is summarized by the function name and innermost loop in which it occurs). E is a set of edges (u,v) such that $u \in U$ denotes an array variable, and $v \in V$ denotes a use of the array. Figure 4.16 shows a small subset of a sample use graph. The left entries (U) denoted by Field corresponds to the arrays specified in the user specification file, along with its type. The right entries (V) specify function names and inner loop statement number. This graph aids in easy identification of common array accesses across the loops. For example, if two arrays

never share a common use, they are not likely candidates for merging. From the sample graph, it is clear that merging of arrays for layouts is a non-trivial problem. For example, arrays x , y and z are exclusively used in `CollectElemPositions` function and jointly used in `LangrangeNodal` function with arrays xd , yd , zd , xdd , ydd and zdd . Merging these two arrays sets would lead to better locality for `LangrangeNodal`, but lead to additional lines being fetched in `CollectElemPositions`. The complexity of automated layout selection increases with larger numbers of arrays and array references. We believe that the use graph is a useful data structure for guiding the selection of data layouts, whether the selection is done automatically or manually.

4.3.2 Cache-Use Factor(CUF)

We introduce the cache-use factor as a metric to capture the possible cache impact of merging two or more array groups. We explicitly use dynamic loop counts from profiled data along with the use graph to calculate this metric. This factor indicates usage efficiency of cache lines fetched during a loop execution. On merging two arrays groups, the cache-use factor may be lowered since all arrays in both groups might not be used across the loops. For example in Figure 4.16, merging array groups $\{x,y,z\}$ and $\{xdd,ydd,zdd\}$ will likely lead to a low cache-use factor, since these groups just share a single loop `LangrangeNodal` in common.

Before defining cache-use factor, we define another term cache-loop factor (CLF). CLF metric is weighted loop count values for a given loop, where weight is a fractional value

between 0 and 1, calculated based on arrays present in a given loop. If all arrays in an array group are used in a given loop, CLF is equal to the loop count. The cache-loop factor is defined as follows, given an array group A (i.e. set of arrays) and loop L ,

$$CLF_L = \frac{|\{A\} \cap \{array\ references\ in\ L\}|}{|A|} * LC_L \quad (4.1)$$

where LC_L denotes the loop count of loop L . The cache-use factor is then defined as follows,

$$CUF = \frac{\sum_{i=1}^{numLoops} CLF_i}{\sum_{i=1}^{numLoops} LC_i}, \text{ where } LC_i = 0 \text{ if } CLF_i = 0 \quad (4.2)$$

For the base case, where all array are separate, we have $CUF = 1$. However, as we merge array groups the CUF value might lie somewhere between 0 and 1 (inclusive of the two bounds), with 1 indicating better cache line use. This might lead us to refrain from merging some arrays, as we desire the highest CUF. However, whenever possible, we merge array groups as it helps in better register use, prefetching and locality of elements as we have seen in Section 4.2.

4.3.3 Automatic Data Layout Algorithm

Our automated data layout algorithm uses the cache-use factors and platform characteristics to produce a meta file that contains the recommended data layout. Algorithm 1 shows the automated data layout algorithm. To begin, each array in the field specification is placed in its own ArrayGroup. The algorithm compares all pairs of ArrayGroups to determine the

Algorithm 1 Automated Data Layout Algorithm

```

1: procedure AUTODATALAYOUT(ArrayGroupList)
2:   while IsMerge is true do
3:     IsMerge  $\leftarrow$  false
4:     for  $pairs \in ArrayGroupList$  do
5:       if (pair writes)  $>$  2*(pair reads+pair read/writes)
6:         Ignore pair
7:       end if
8:       best pair  $\leftarrow$  pair with highest cache use factor
9:     end for
10:    if best pair  $>$  threshold
11:      merge pair
12:      IsMerge  $\leftarrow$  true
13:    end if
14:  end while
15:  sortGroups(ArrayGroupList)
16:  splitCacheLine(ArrayGroupList)
17:  return ArrayGroupList
18: end procedure

```

profitability of merging each pair. The pair with the highest cache-use factor is merged to form a new group. This process is repeated until the best candidate pair for merging falls below the acceptable merge threshold. After the final grouping is determined, each group's arrays are sorted based on data type (largest data size to smallest data size), to better pack them. The final step performs cache line splitting i.e. split array groups based on cache line boundaries of an architecture, to efficiently utilize each cache line fetch for the target platform.

The evaluation of the profitability of merging two candidate ArrayGroups considers two factors. The first consideration examines reads versus writes to an ArrayGroup. Our manual results (Section 4.2) showed that grouping arrays written to frequently with arrays

Benchmark	Power7- 8Threads	AMD APU- 4Threads	Sandy Bridge- 8Threads	BG/Q- 64Threads
IRSmk - Best Manual Layout	4.70	1.46	1.11	2.20
IRSmk - Automated Layout	4.67	1.43	1.10	2.08
LULESH - Best Manual Layout	1.43	1.50	1.02	1.10
LULESH - Automated Layout	1.58	1.46	0.96	1.07
SRAD - Best Manual Layout	1.35	3.13	1.00	1.08
SRAD - Automated Layout	1.20	2.55	0.46	0.98

Table 4.3 : Speedup of best manual layout and automated layout speedup relative to base layout

that are only read can decrease performance significantly. Our current heuristic prohibits creating a new merged ArrayGroup, if the number of write-only arrays is more than $2 \times$ the number of read and read-write arrays. The second consideration for merging ArrayGroups computes the cache use factor for the proposed combination. If the cache use factor is greater than our established thresholds, the ArrayGroups are viable for merging. From our empirical results, we have chosen *Cache Use threshold* = 0.57 for our algorithm.

4.4 Automatic Data Layout Results

Table 4.3 shows the speedup of the best manual layout and the automated layout relative to the base layout. The results demonstrate that automated layouts were close to 95-99% of best manual layout for IRSmk, and close to 90% of best manual layout for LULESH

except on the Power 7 where it performs better than the manual layouts on 8 threads. These results prove the effectiveness of our automated results. In one particular case, 8 Threads on Power7 for LULESH, automated layout improved performance as compared to manual layouts. For SRAD, automated results were close to 88% for Power 7 and BG/Q. However, on AMD APU and Sandy Bridge, automated results could not match the performance of user specified layouts. On AMD APU, we speculate compiler specific optimization improving performance on the original benchmark by observing compiler debug information. Incorporating some hardware counter profiling in our algorithm would help in selecting better layouts. However, hardware profiling is not considered in this work. On Sandy Bridge, Intel compiler prohibited vectorization on layout optimized code as mentioned in Section 4.2.1, thereby degrading performance of automated layouts. One could add a prepass to ignore layouts where backend compiler would efficiently vectorize the original source code.

Another point to consider is that these architectures exhibit NUMA behavior, which our automated algorithm doesn't consider for this work. We believe that either extending our algorithm to incorporate memory allocation or using NUMA libraries for memory allocation would further increase layout performance on these architectures. However, in this dissertation, we only used the default memory allocation provided on these systems and leave NUMA enabled data layouts for future work.

Version	Register Spills
Base	348
Layout 9	220
Layout 11	153
Layout 12	159

Table 4.4 : Register spills for IRSmk on AMD APU for three different layouts

4.5 Performance Analysis

This section analyzes the performance gains obtained by our automatic algorithm, as an extension to the performance analyses reported in Section 4.2. We conducted extensive experimentation to determine that performance is impacted by three factors: register allocation, prefetch streams and locality. In this section, we analyze each of these effects.

4.5.1 Register Allocation Analysis

To observe register allocation efficiency, we measured register spills generated for the base code in comparison to different layouts. We had to modify the internal register pass in *gcc* for this purpose. We show limited results in this section due to space limitations, but have confirmed that these conclusions hold across all the layouts and benchmarks that we studied.

Table 4.4 shows the effects of register allocation across different layouts for IRSmk on *AMD APU*. These results show that there is significant decrease in register spills (upto $2.28\times$) due to the layout transformation. On close observation, we found that in the Base version, the code generation pass tried to allocate a single register for every array variable,

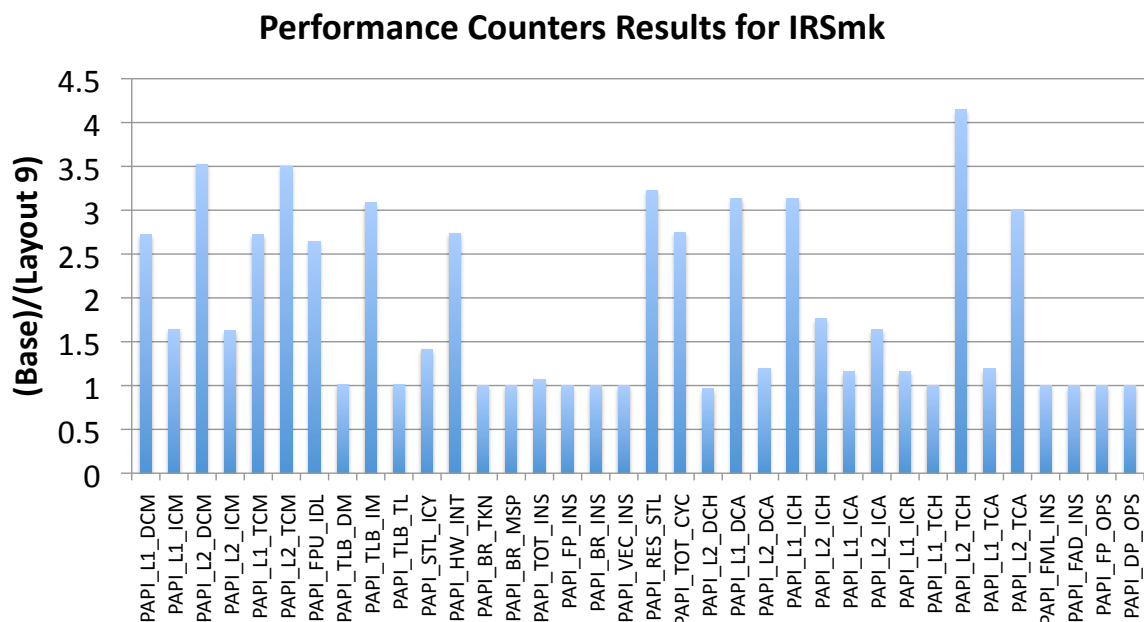


Figure 4.17 : Hardware performance counter results for IRSmk on AMD APU

thereby increasing register pressure. However, the layout transformation greatly impacted this behavior. As we used array of structs for our transformation, the code generation pass can allocate a single register for every array of structs. This effect reduced the number of register to be simultaneously hold for the same benchmark, leading to reduced spills. Though register spills has a great impact on performance, merging all the arrays into a single AoS does not lead to the most efficient code. Locality and prefetch streams also impact performance, as discussed next.

4.5.2 Locality and Prefetch Streams

Layout transformations have deep impact on spatial locality due to the organization of data within a cache line. Figure 4.17 shows hardware performance counter results collected

for IRSmk on *AMD APU* machine using the PAPI Interface. Results show the ratio of each of the performance counters for base relative to Layout 9. Similar behavior was observed across the layouts and different benchmarks. From the overall results, we find that the data cache misses (L1_DCM and L2_DCM) have decreased significantly showing improvements of $2.5\times$ and $3.5\times$ respectively. This effect can be attributed to the better cache management by our layout transformations. Improving locality not only impacts performance but also reduces the stalls (RES_STL) in the program. An interesting thing to observe is the decrease in L1 data cache accesses (L1_DCA) and instruction misses. Due to better register allocation, there are fewer accesses to the memory hierarchy, leading to reductions in L1 accesses and instruction misses. Overall, we observe significant improvements across most of the performance counters.

From our analysis, we also found that prefetch streams improved by using the layout transformation. The reason being that most of the current architectures have limited prefetch streams which fetch arrays accesses from the code. When arrays are combined into array of structs, the number of prefetch streams reduces by the number of arrays present in the AoS. This effect helps in improving the overall performance of an application as the pressure on prefetch streams reduces as we merge arrays into AoS.

4.6 Summary

In this dissertation, we establish the foundation for a new approach to supporting portable performance of scientific codes across HPC platforms. The upgraded TALC

source-to-source transformation tool permits application developers to maintain one “neutral” data layout source code and explore architecture specific array layouts. The new automated portion of TALC can analyze the original source code based on platform characteristics and produces a new source code with new array data layouts ready to be compiled and run on that system. The results for the three test codes show that manual layouts improve performance by $1.10\times$ to $22.23\times$ for IRSmk, $1.00\times$ to $3.68\times$ for SRAD and $1.02\times$ to $1.82\times$ for LULESH with results varying with thread count and architecture. The automated algorithm resulted in performance of 95-99% of the best layout manual layout for IRSmk. For LULESH, the automated results was close to 90% of the best manual layout on all other processors. For SRAD, automated results were close to 78% of the best manual layout for all architectures except for Intel Sandy Bridge where layouts interfered with vectorization provided by Intel compiler.

Chapter 5

Automatic Selection of Distribution Function

Distributed applications use multiple cluster nodes to run an application. These applications allocate multiple sub-tasks onto different cluster nodes during its execution. Programming models often expose distribution choices of sub-tasks across cluster nodes to a programmer for efficient computation. Programmers are faced with challenges such as communication pattern, load balancing, task dependencies and locality optimization to decide efficient distribution of sub-tasks. In the past, distribution functions such as block, cyclic and block-cyclic have been commonly used in the literature. However, experimenting with such distributions takes multiple runs to tune an application due to parameters associated with each of distributions.

In this work, we develop a novel model to automatically select a distribution function for an application. Our approach uses dynamic graph generated from an application run and creates an analytical model on this graph to compare different distributions and select an efficient one out of them. To demonstrate the effectiveness of our approach, we use Intel CnC programming model. Intel CnC model provides an efficient way to specify distribution function for allocating tasks and data across different cluster nodes. In the next section, we provide details about the Intel CnC programming model and its distributed behavior. Section 5.2 discusses Cholesky benchmark and its distribution across nodes. In

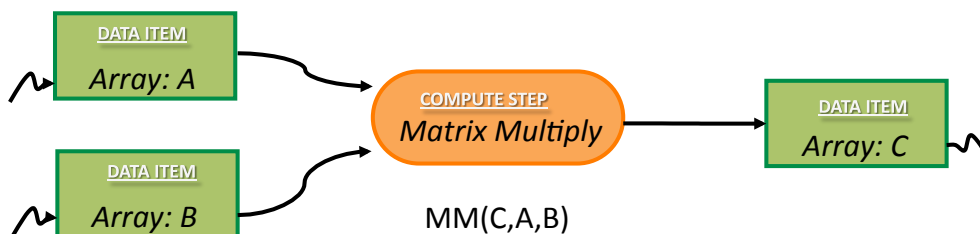


Figure 5.1 : Matrix multiplication example in CnC. Note, this is a very simplistic example of matrix multiply in Intel CnC. Other implementations with finer granularity will be more efficient

Section 5.3, we present our framework to automatically select a distribution function. Our experimental results are presented in Section 5.4, followed by summary in Section 5.5.

5.1 Intel CnC Programming Model

Intel CnC Programming Model [16, 17, 87] is a generic programming model, which exposes data and control dependence to a programmer using dynamic single assignment semantics *. The Intel CnC programming model is deterministic in nature and is well adapted for asynchronous parallel applications. There are three main constructs in the Intel CnC model:

- Step Collections

Step Collections corresponds to basic units of executions in an application. In Figure 5.1, matrix multiply operation is a step collection. Step Collections are represented

*Dynamic Single Assignment property states that there will be only one writer for a data item. The data item can be read multiple times

by capsule like shape.

- Data Collections

Data Collections represent set of data items in an application. These are data that is produced (using put operation) or consumed by (using get operation) steps. However, due to dynamic single assignment property, a data item is produced only once but may be consumed many times. In Figure 5.1, A, B and C are data collections. Rectangular boxes represent Data Collections. In some references, data collections may be referred to as item collections.

- Control Collections

Control Collections corresponds to factory of step instances [17]. These are used as control inputs to determine which step instances will execute. In the programming model, the term control tag is associated with control collections. A step collection will be associated with a control collection, which executes only if a matching tag exists for the corresponding step. However, details like when a step should be executed are left to the CnC scheduler.

As mentioned earlier, tags are associated with control collections. However, tags are also associated with step collection and data collection instances to uniquely identify them. Steps may produce tags to control other steps, which ensure an ordering between them. As shown in Figure 5.1, edges show the flow of data into a step and from it. The curved arrows represent the environment, which means an item or tag comes from or arrives into

an environment. These environment actions are necessary for initializing a data item or tag and producing output from an application.

Besides the constructs, Intel CnC imposes two sources of ordering constraint in their model for an application. First, a producer step must execute before the consumer step. This ordering is fundamental, as the consumer step needs to consume an item produced by a producer. The second ordering specifies that controller must execute before a controllee. This ensures that a step producing a control tag always executes before a step consuming the tag. More details about Intel CnC programming model can be found on the website [16].

Distributed Intel CnC

The simplistic producer-consumer relationship model in Intel CnC makes it a viable candidate for distributed memory applications. Steps and data can be easily distributed across a cluster for efficient computations. However, communication across cluster nodes is necessary for transferring data or control tags between steps. Also, the dynamic single assignment feature in Intel CnC helps in maintaining consistent data across cluster nodes.

As we create distributed applications, an important point to consider is how should we distribute data and tasks across the nodes. In the CnC model, programmers have the flexibility to choose both of these options, i.e. distribute task and/or data in an efficient way. Programmers can specify tuning options to control distribution of computation units or specify where a particular data item resides.

The default implementation is a round-robin distribution of computation units across

the cluster nodes. However, round-robin distribution may not always be an efficient choice to distribute tasks. Experimental results for four benchmarks: cholesky, matrix inverse, primes and mandelbrot [†], have demonstrated performance losses for default distribution in [88].

In the Intel CnC model, programmers can specify a user-defined distribution function in a separate tuning file to improve from the default behavior. A separate tuning file is always desired as it enables separation of concerns from problem at hand with performance tuning. Note that the distribution function in the tuning file is devoid of any low level constructs such as specifying explicit communication among tasks or data. Communication and shipping of data is automatically handled by the runtime. Following are three tuning functions to enable efficient distribution, which can be used by the programmer:

- `compute_on`

This method enables task-based distribution. Programmer may specify user-based distribution in this method to appropriately schedule steps across cluster nodes by Intel CnC runtime.

- `consumed_on`

This method enables efficient communication by specifying which cluster nodes will consume the data items produced by the current step. If this method is not present, Intel CnC runtime will broadcast the data item to all the cluster nodes. This method provides an efficient way to distribute the data items.

[†]Benchmarks are available in standard Intel CnC package [16]

- `produced_on`

This method is substitute to `consumed_on` method. This method will be only evaluated if the `consumed_on` cannot determine where data items will be consumed. `produced_on` specifies where a particular data item will be produced to enable efficient communication in the Intel CnC runtime system. Thus, a step will remotely fetch a data item from another cluster node appropriately.

In this work, we restrict ourselves to selecting an appropriate distribution function for tasks. However, the approach mentioned in this proposal is in no way restricted to only task based distribution and can be easily adapted to data distribution as well. We use `compute_on` function to assign tasks across different cluster nodes. We also employ `consumed_on` function to determine the nodes where the items produced by a step should be shipped. `consumed_on` function helps in generating lesser message traffic across the cluster nodes by directing items produced to where it would be needed in the future.

5.2 Distributed Cholesky Example

Cholesky benchmark is used for solving system of linear equations of the form $Ax = b$. Cholesky method solves these equations by using a decomposition step which solves $A = LL^T$, where L is lower triangular matrix and then uses substitution method to calculate values for x . Cholesky decomposition is a time consuming step and often results in overall bottleneck for the benchmark. Various scientific applications use cholesky decomposition and it is an important application in the Intel CnC framework. Thus, speeding up Cholesky

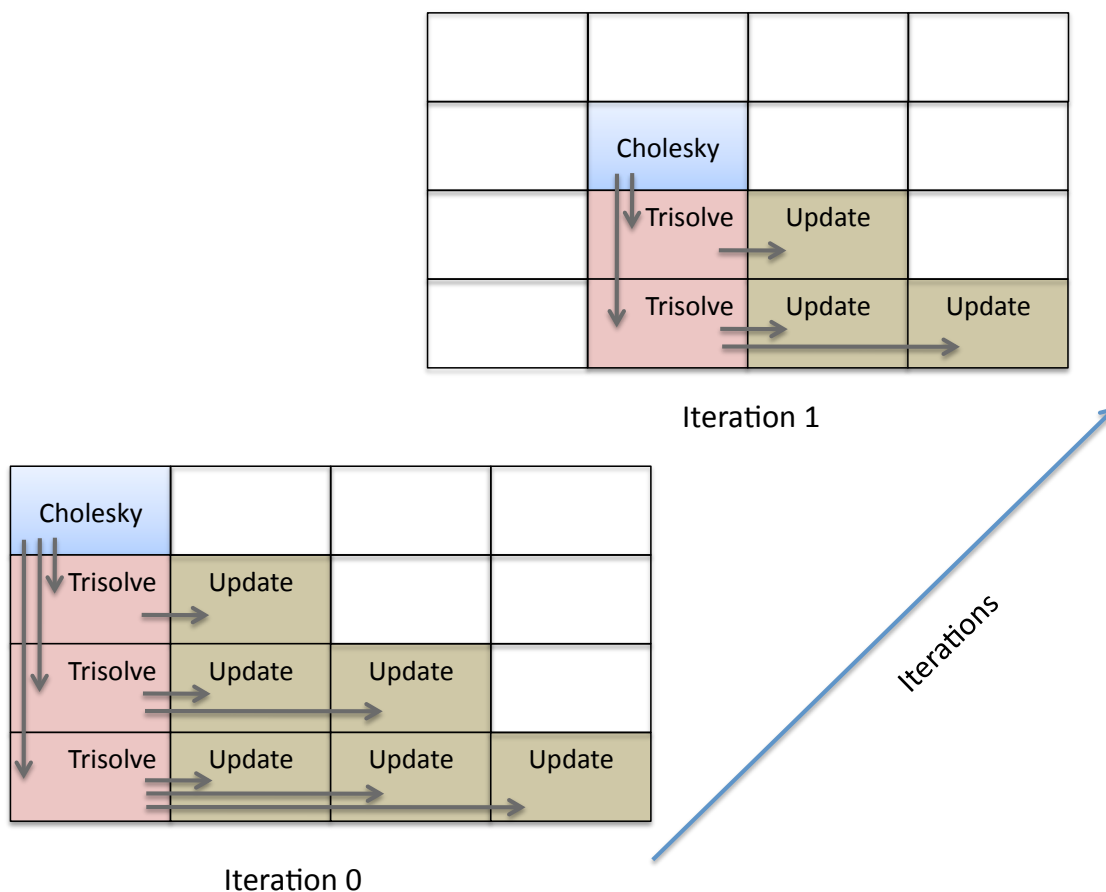


Figure 5.2 : Execution steps of Cholesky benchmark

benchmark is important.

For faster execution, it is a good option to run the Cholesky decomposition in parallel. Using parallelization, multiple steps can be computed at the same time. Past work have demonstrated the benefits of parallelizing cholesky benchmark [89,90,91]. An important factor for cholesky performance is communication across the different computations.

Figure 5.2 shows the overall execution steps of Cholesky decomposition in Intel CnC. Cholesky decomposition is divided into three distinct steps: Cholesky, Trisolve and Update,

which work on separate parts of the data matrix in each iteration. Trisolve step computes corresponding column values for each Cholesky step, whereas update step computes row values to the corresponding Trisolve step. The arrows in the figure indicate the dependency across the different steps. For example, Trisolve is dependent on Cholesky step. Iterations in the figure, compute different elements of the matrix. Dependencies across the iterations are not shown in this figure, but it can be easily deduced from the different steps across the matrix in the same position. Each block represents a tile or block of operation for a step. Computing tasks in blocks is beneficial as it balances the computation to communication ratio. Intel CnC provides an efficient method to model the Cholesky decomposition. Each step is modeled in terms of its dependencies. As soon as, all the dependencies of a step are satisfied the runtime schedules a step for execution. On a multi-core chip, the Intel CnC runtime automatically places these steps based on Intel TBB scheduling. Employing a distributed multi-node cluster is a feasible option as we scale out the Cholesky benchmark across multiple nodes.

Figure 5.3 illustrates working of Cholesky benchmark across distributed cluster nodes. A key difference between the earlier figure (Figure 5.2) and this figure is that the Cholesky application is run across multiple nodes. In this figure, each box associated with a step which indicates the cluster node on which the computation takes place. For example, Cholesky step in iteration 0 computes on node 0. In this particular example, steps were distributed across four nodes (Node 0-3). Applications can use more nodes depending on the distributed environment. Intel CnC framework provides an extensible model for

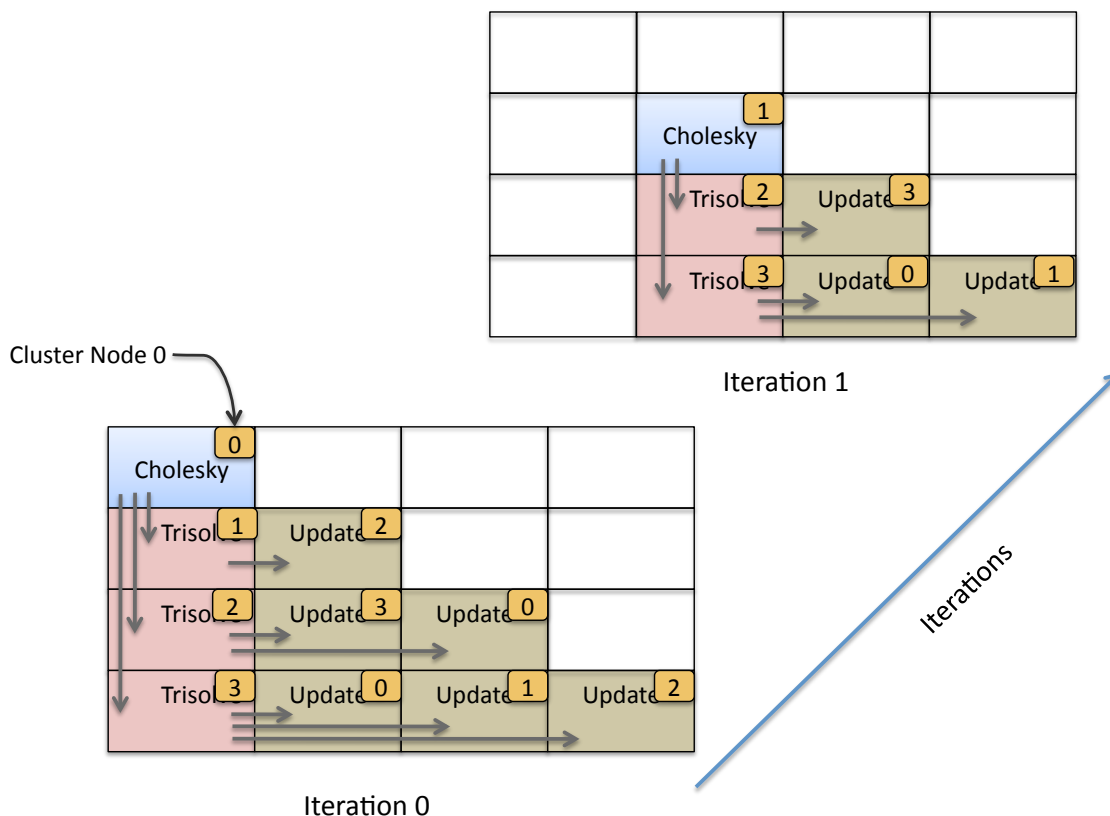


Figure 5.3 : Cholesky benchmark on distributed cluster nodes

scaling applications across different nodes. The arrows in this example also shows the dependencies across the different steps. However, if steps are computed on different nodes, communication takes place across the different nodes to transfer data computed from one step to another. If steps are computed on the same node, intra-node communication occurs which is less expensive than across the nodes. In this example, we have shown one possible distribution of steps across the cluster nodes. Intel CnC provides with a tuning interface where a programmer specifies different distributions. Based on these task distributions, overall execution of the application is impacted as distributions lead to different

computation and communication behavior.

5.3 Distribution Function Selection Model

Intel CnC programming model provides an efficient way to specify distribution function. Programmers can choose to specify their own distribution or use any of the standard distribution such as blocked, cyclic or block-cyclic. These distributions can be specified using the three tuning functions, specifically `compute_on` for task distribution. However, these choices lead to experimenting with multiple runs for an application to select the best distribution. Distributions such as block-cyclic need multiple parameters to control the number of elements distributed across the cluster nodes. Programmers may often limit their choice by selecting a sub-optimal distribution when full-scale experimentation is not possible. In this section, we describe an automatic distribution function selection approach. Our approach builds on a dynamic graph analysis of CnC application with analytical model to choose parameters for linear regression. First, we describe our approach to compute different parameters, which help in distinguishing different distributions. Then, we present our linear regression model to select the best distribution for an application.

5.3.1 Framework for Parameter Generation

One of the challenges that programmers face is to select an efficient distribution function. An automatic model should be able to compare different distribution to present a final efficient solution. In order to distinguish different distribution choices, we use varying pa-

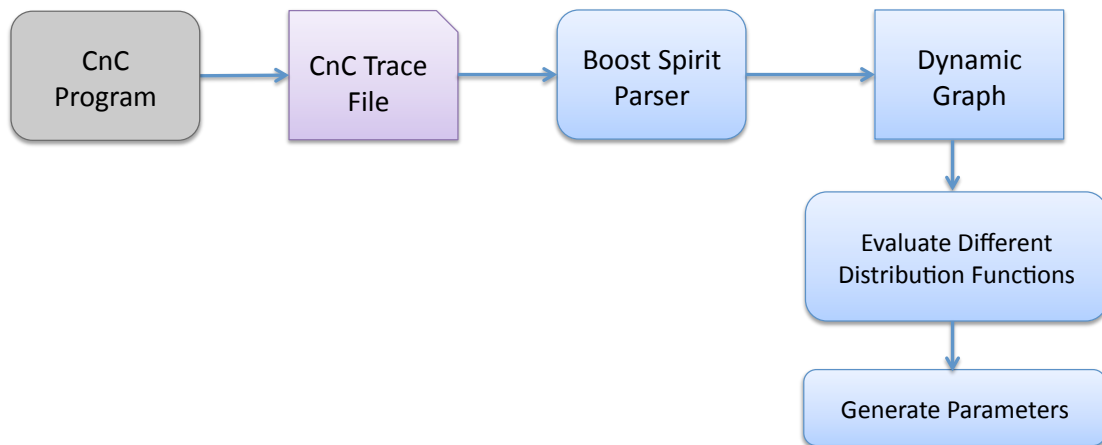


Figure 5.4 : Framework for parameter generation

parameters to compare them. Appropriate parameter selection is key to creating an effective overall model. These parameters act as input to our linear prediction model, which automatically selects an efficient distribution. In this section, we describe our framework for generating different parameters using an analytical model. A key factor to consider in this approach is that the parameters are generated using a single sequential run of the application. Parameters are produced by mapping the distribution choices on a dynamic graph. This approach helps in avoiding expensive runs across all the possible distribution function choices. For now, we just describe the framework and in Section 5.3.2 present each of these parameters.

Figure 5.4 shows our approach to collect different parameters across the distributions. First, we collect CnC trace through a sequential run of the application. Tracing is essential in our approach since it gathers dynamic dependencies and association between steps


```

Put item [Lkji: [ 0, 0, 0 ]] -> UnformattedType at 0x2aaaabe00bf0
Put item [Lkji: [ 0, 1, 0 ]] -> UnformattedType at 0x2aaaabe00c50
Put item [Lkji: [ 0, 1, 1 ]] -> UnformattedType at 0x2aaaabe00ce0
Start step (Cholesky: 0)0x2aaaabcf3f20
Get item [Lkji: [ 0, 0, 0 ]] -> UnformattedType at 0x7fffffc000
Put item [Lkji: [ 1, 0, 0 ]] -> UnformattedType at 0x2aaaabe00d70
End step (Cholesky: 0)
Start step (Trisolve: (0,1))0x2aaaabcf3f20
Get item [Lkji: [ 0, 1, 0 ]] -> UnformattedType at 0x7fffffbff0
Get item [Lkji: [ 1, 0, 0 ]] -> UnformattedType at 0x7fffffbfe0
Put item [Lkji: [ 1, 1, 0 ]] -> UnformattedType at 0x2aaaabe00e00
End step (Trisolve: (0,1))
Start step (Cholesky: 1)0x2aaaabcf3e40
Get item [Lkji: [ 1, 1, 1 ]] ( not ready )
Suspend step (Cholesky: 1) (input item not ready) 0
Start step (Update: [ 0, 1, 1 ])0x2aaaabcf3dd0
....

```

Figure 5.5 : Sample CnC Trace for Cholesky benchmark

and items. Intel CnC model provides a clean tracing framework with detailed information about step executions and data item uses within a step. Figure 5.5 shows a sample trace collected for Cholesky benchmark. Note that put item calls are data items produced by current step and get item denotes data items consumed by a step. Start step and end step implies beginning and termination of step execution. The hex code shown in figure is used for internal purposes in the Intel CnC environment and can be safely ignored for this work. In the figure, Lkji is a data item collection and Cholesky and Trisolve are steps with their tags.

The trace file helps in understanding the relationship across steps through data items. Next, we parse the trace file to create a dynamic graph for our analysis in distribution function selection. Our parsing framework is built on the standard boost spirit framework

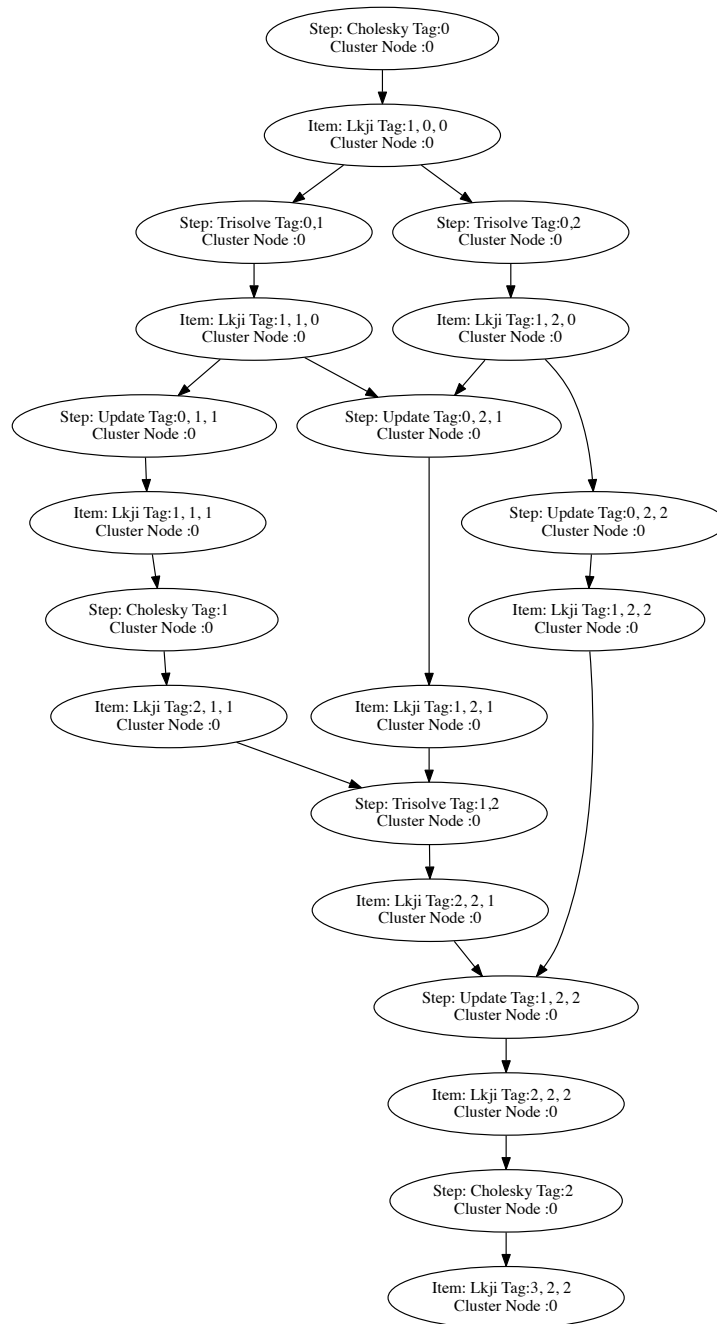


Figure 5.6 : Dynamic graph for Cholesky benchmark

[92]. Figure 5.6 shows dynamic graph for Cholesky example. Steps denote dynamic steps in the application and item denotes the data item produced and consumed across different steps. In this example, cluster node value assigned to each item and step represents that all steps and items were computed on the same cluster node. Our goal is to automatically distribute these steps and data items across cluster nodes for efficient computation.

After generating the dynamic graph, we proceed to the next step of evaluating a distribution function for a given application as seen in Figure 5.4. First, we assign cluster nodes to different steps and items based on a distribution function and then generate different parameters to compare these distributions. These parameters can be considered as a goodness of fit test to judge different distributions. Although this approach evaluates multiple distributions, it does not perform multiple dynamic runs for the application. Our approach evaluates distribution function using the generated dynamic graph. This helps in limiting the number of runs for any application.

5.3.2 Parameter List

Here, we present the parameter list which helps the linear regression in predicting an efficient distribution.

1. Critical Path Length

This parameter denotes the critical path length of the application using the dynamic graph. Critical path is the longest directed path from the start node in the dynamic graph to the final node.

2. Number of Communication Links

This parameter represents the number of communication links present in the dynamic graph. For example, if a Cholesky step is computed on Node 0 and a Trisolve step using its data, is computed on Node 1, then we count a single communication link. However, if both of them were computed on the same node, we would avoid counting local communication.

3. Total Communication

In the Intel CnC model, certain tags are communicated from master node to other nodes during the execution. The key difference between this parameter and earlier one (Number of communication links), is that this parameter counts the total communication links present in the application. In contrast, number of communication links parameter only add communications present in the dynamic graph of an application.

4. Critical Steps Ratio

This ratio denotes the number of different steps present on the critical path to the total number of steps in the application.

5. Step Load Balance Factor

This parameter signifies the load balancing factor for different steps present in an application. The load balancing factor is calculated as the standard deviation across the number of steps distributed on different cluster nodes for a given distribution function. For example, if the total number of Cholesky steps computed on Node 0

and 1 are 12 and 13 correspondingly, we will have a good load balance factor as compared to 20 and 6 steps were computed on the same nodes. Lower value of load balance factor denotes a better load management across the nodes. Ideal value of load balance factor should be close to 0, indicating equal balance across all the nodes.

5.3.3 Overall Model using Linear Regression

Linear regression is a modeling technique where data is modeled using linear predictor functions. Unknown model parameters are learned from the data and applied to future observations. Overall, linear regression models the relationship between variables to response by fitting a linear equation to observed data. If linear regression has a single variable present, then the model is termed as *simple regression*. A different model, *multiple linear regression* (MLR) is used for determining relationship between two or more variables corresponding to an observed response. Linear regression models have long been used in various domains such as finance, economics and statistics.

We model the selection of distribution function using *multiple linear regression* (MLR). From this point onwards, whenever we refer to linear regression, it denotes MLR. Linear regression model was a good fit for our problem statement, since we had to choose a minimum execution time from a set of search space points based on a few parameters. We found that the model parameters could be easily learned from a small set of observations and later be applied to the entire search space to find the best point (minimum execution time). Based on these facts, we chose linear regression to map the distribution selection

problem.

The linear regression model is represented as follows,

$$y_i = \beta_0 + \beta_1 x_{i,1} + \dots + \beta_n x_{i,n} + \xi_i, \text{ where } 0 < i \leq N$$

In the above equation, $x_{i,1}, \dots, x_{i,n}$ are N predictor variables and y_i is the corresponding i^{th} response. N is the total number of observations for training data. β 's are the corresponding linear model coefficients which are learned from training or sample data. ξ_i denotes the errors in the model and is used to adjust the overall linear regression model. This equation is a generalized MLR equation. We need to appropriately set the response and predictor variables for our problem statement.

The above equation is adapted for distribution function selection model. For our model, y_i denotes the overall execution of a single distribution function run and x_i 's represent the set of variable parameters as presented in Section 5.3.2. β and ξ is learned from the training set or sample data.

Figure 5.7 shows the overall flow of our model using linear regression. First, we sample random data points from the overall search space of distribution function choices. We perform a stratified random sampling to have equal representation from the different distribution functions. These data points are used as training data for our MLR model. Sufficient points are collected to avoid under-fitting or over-fitting the regression model. As the next step (Step 2), we generate the different parameters using the framework described in Section 5.3.1. We also execute these different distribution choices on the real environment to

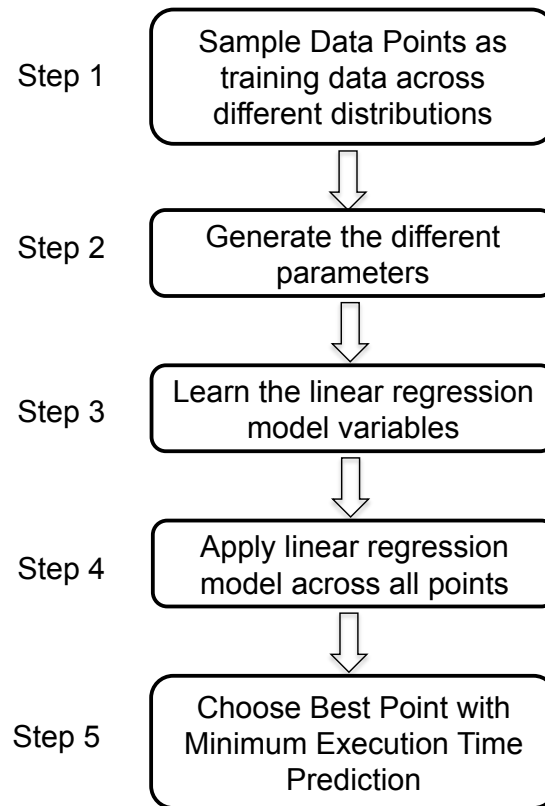


Figure 5.7 : Overall steps in Linear Regression Model for distribution function selection.

note the response (y_i). In Step 3, we use these parameters to learn the overall regression model. The x_i values are set of parameters generated in the previous step and y_i is execution time. This model results in generating the β and ξ values. Learning the model helps in predicting the execution time for any choice of distribution function.

By using this MLR model, we predict execution time across the whole set of data points in the search space in Step 4. Note, that this search space is sufficiently larger than the training data. Thus, we need to run the application for a limited set of points (training data) rather than the entire search space. We use β and ξ values from the previous step

and x_i are generated from the parameter framework without running the program itself. This approach helps in predicting the execution time for different data points and select the minimum execution time from the entire search space in Step 5. Our method avoids expensive runs for the entire search space and at the same time, is not overly simplistic by using only analytical methods to predict best distribution function choices for a user. All of these steps are automatically executed without little or no user intervention. This approach is not restricted to Intel CnC model, but can applied to a much broader generalized framework.

5.4 Experimental Results

For our experiments, we used an 8-node Intel Westmere (X5660) based cluster. Each node is configured with two Intel Westmere chips having 6-cores per chip (12 cores per node) running at 2.80 GHz. A single core has 32KB L1 data cache, 32KB instruction cache and 256KB L2 cache. Each chip contains a 128MB shared L3 cache across the six cores. The total main memory capacity of each node is 48GB (4GB per core). All the nodes are interconnected using QDR Infiniband network with capacity upto 40 Gb/sec. We used Intel CnC v0.9 with Intel MPI and gcc v4.6.4 with -O3 optimization level across all our experiments. For all applications, we use 12 threads per node (1 thread per core).

We conducted experimented on Cholesky benchmark available in the standard Intel CnC distribution. We selected a problem size of $N=5000$ and block size (BS) = 50 for Cholesky. Note, that block size parameter is a tile of computation performed by each step

Steps	Block-Cyclic1 (Distb1)	Block-Cyclic2 (Distb2)	Checked (Distb3)
Cholesky Step (i)	$(i/\text{blk1})\% \text{numNodes}$	$(i/\text{blk1})\% \text{numNodes}$	$(i/\text{blk1})\% \text{numNodes}$
Trisolve Step (i,j)	$(j/\text{blk3})\% \text{numNodes}$	$(i/\text{blk2})\% \text{numNodes}$	$(i/\text{blk2} + j/\text{blk3})\% \text{numNodes}$
Update Step (i,j,k)	$(j/\text{blk5})\% \text{numNodes}$	$(i/\text{blk4})\% \text{numNodes}$	$(i/\text{blk4} + j/\text{blk5} + k/\text{blk6})\% \text{numNodes}$

Table 5.1 : Different distribution functions. blk denotes blocks and numNodes represents number of cluster nodes in distributed environment.

in the benchmark. This parameter is different from any of distribution function parameter.

Three different distribution functions along with varying parameters were chosen for our experiments. The three distribution functions were based on two variations of block-cyclic and a checkered distribution. Table 5.1 shows the distribution function for each of the steps in Cholesky application. There are six block parameters across the three steps. Each of these block parameters control the number of tiles that would be placed on the same nodes in each dimension. As these parameters could vary from 0 to dimension itself, we sub-sampled the search space to limit our processing time. Block sizes were selected from the following set of numbers: [1 2 5 8 10 12 15 20]. Explicit enumeration of these block sizes across different steps provided sufficient data points. These sampled points were sufficient to create appropriate representation of the entire search space. Based on these parameter selections, we had 512 data points for each distribution and 1536 points overall. In the next section, we present performance variations across these data points, followed by our linear regression model results.

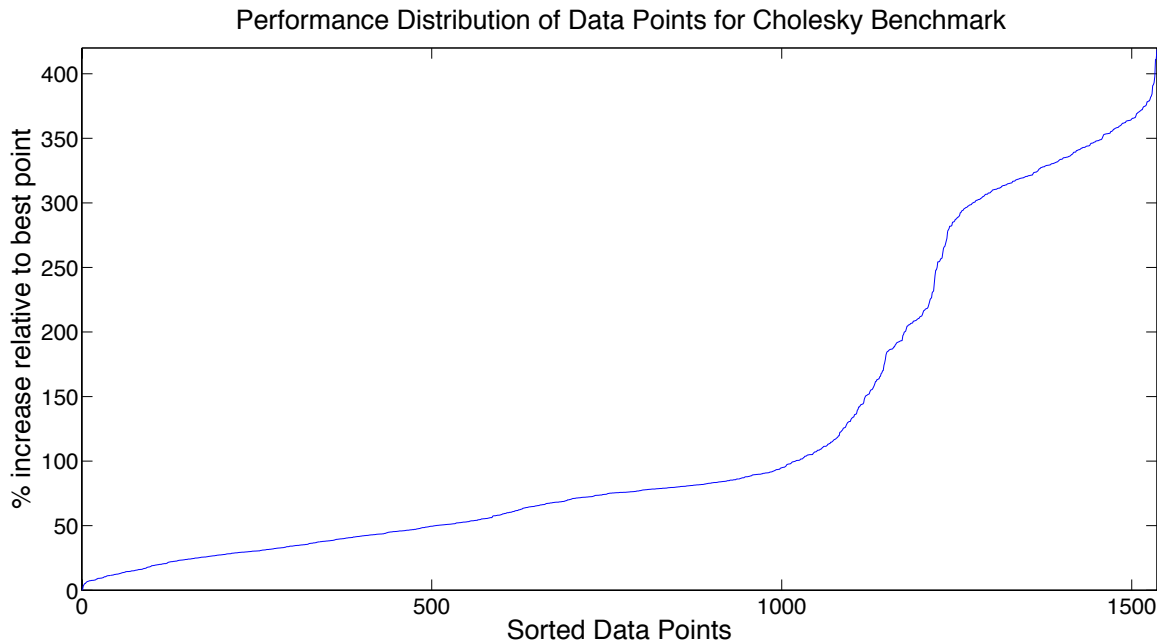


Figure 5.8 : Cholesky performance variation across the different distribution functions. x-axis shows the sorted data points based on execution time. y-axis shows the percentage increase in execution time compared to the best point. Best point refers to the minimum execution time across the data points.

5.4.1 Performance Variation across Different Distribution Functions

Figure 5.8 shows the performance variation across the set of data points for Cholesky benchmark. A single (x,y) point in this graph shows the relative performance of that data point compared to the best point (minimum execution time). The data points are sorted based on execution time. Thus, the minimum execution time appears in the lower left corner whereas the highest execution time (worst point) appears on the top right corner. From the figure, we clearly observe that there is significant performance variation across the data points, which shows that choosing the right distribution with correct parameters is important to have efficient application performance. The performance variation between

Model	Predicted Actual Execution Time (secs)	Overall Best Execution Time (secs)	% Difference
Linear Regression Model applied across search points	0.8428	0.7222	16.69%

Table 5.2 : Linear regression model results for Cholesky benchmark across all the data points.

the best point to worst point is from 0% to 418%.

From the results, we observe that only about 6.50% data points are no more than 18% slower than the best point. This shows the complexity in choosing a data point close to the best point is a difficult decision. A efficient distribution selection model would pick a point that would be closest to the best data point. In the ideal case, this data point would be the minimum data point.

5.4.2 Linear Regression Model Results

Results presented in this section is based on our linear regression model in Section 5.3.3. We use stratified random sampling of 150 data points as training data. For Cholesky application parameters, we use Critical Path Length, Number of Communication Links, Total Communication, Cholesky Critical Step Ratio, Trisolve Critical Step Ratio, Update Critical Step Ratio, Cholesky Load Balance Factor, Trisolve Load Balance Factor and Update Load Balance Factor.

Table 5.2 shows the results for our linear regression model applied on Cholesky bench-

mark across the search space points. Predicted Actual Execution Time is the real world execution time based on the predicted distribution function by the linear regression model. Overall Best Execution Time denotes the best (minimum) execution time across all the data points. The last column represents the percentage difference between predicted and best execution time. From the results, we observe that the linear regression model was predicted efficient distribution function, close to the overall points. As seen in Section 5.4.1, the model could have predicted a span from 0% to 418%. These results show that the model was efficient enough to predict a suitable distribution function without exhaustively running all the data points in this search space.

5.5 Summary

Distribution function play a key role in determining the performance of an application. Complexity of this problem further increases as programmers have not only to choose an efficient distribution function, but also the correct parameters associated with it. In this dissertation, we developed a novel approach that helps solve this problem by using automatic selection model based on linear regression. The linear regression quickly learns the parameters based on a few sample inputs and then predicts an efficient distribution function. We demonstrate the effectiveness of this approach on Cholesky benchmark using the Intel CnC platform. Our results indicate that linear regression model based approach can predict efficient distribution function which had a difference of 16% from the optimal point. We believe that our model shows promising approach to select an efficient distribution.

Chapter 6

Related Work

Locality optimization improves performance of an application by reusing cache elements in the memory hierarchy. As stated earlier, our focus in this work, for the first two problems is tile size selection and data layout optimization, both of which are methods of locality optimization. For the third problem, we look at automatically selecting a distribution function for inter-node performance. There has been a lot of prior work in all of these domains. In this chapter, we look at past work in tile size selection domain. Next, we explore related work in data layout optimization and distribution function selection. Contents of tile size selection and data layout optimization sections are extracts from our previously published work in [1, 2].

6.1 Tile Size Selection

Exploiting data locality is a key issue in achieving high levels of performance and tiling has been widely used to improve data locality in loop nests. Nevertheless, the choice of tile sizes greatly influences the realized performance. Wolf and Lam [30] were the first to provide precise definitions of reuse and locality and develop transformations to improve locality. Ferrante et al. [64], Wolf and Lam [30], and Bodin et al. [93] were among the ear-

liest to develop cache estimation techniques designed for data locality optimizations. Several authors proposed techniques for selecting tile sizes aimed at reducing self-interference misses [70, 46, 49]. Ghosh et al. [50] developed cache miss equations to find sizes of the largest tiles that eliminate self-interference, while fitting in cache. Chame and Moon [48] developed techniques to minimize the sum of the capacity and cross-interference misses while avoiding self-interference misses. Rivera and Tseng [51] developed padding techniques to reduce interference misses and studied the effect of multi-level caches on data locality optimizations. Hsu and Kremer [47] presented a comprehensive comparative study of tile size selection algorithms. To the best of our knowledge, all of these techniques find a single tile size for each loop that is being tiled. Recently, Yuki et al. [94] have explored the automatic creation of cubic tile size models. In contrast, we demonstrate in Section 3.6 that the best performance is often realized only for rectangular tiles.

Work proposed by Christ et al. [95] and Grey et al. [96] use communication avoidance algorithms for calculating tile size using lower bound analysis on examples such as blocked matrix multiply. Their approach works by using linear programming model based on the affine indices of program references. Their algorithm predicts best tile sizes which minimize communication by using values as large as the constraints fit into the memory. For example, in case of blocked matrix multiply, their approach suggests maximizing the value of block size (b) subject to $3b^2 \leq M$ where M is memory size. In contrast, our approach predicts a range of values, between the DL-ML, as opposed to a single value. For multiple levels of caches, their approach uses recursive decomposition techniques similar to cache

oblivious algorithm to limit the tile sizes at each level. In our model, we have incorporated multi-level cache hierarchies including TLB's in our single level tiling constraints. However, our model in the current form, has not been generalized to distributed applications. In comparison, their approach has been shown to be scalable and effective across single nodes and multi-node cluster environments.

Kondukula et al. [97] proposed a new approach called *data shackling*. *Data shackling* approach applied transformation based on data-centric analysis in contrast to the traditional iteration or control-centric analysis. This work was designed to be useful on imperfectly nested loops. In this approach, the compiler would block the data arrays instead of loops and appropriately schedule statements using this data block. Tile sizes for data was chosen to be fraction of cache capacity, since statements were scheduled whenever data was fetched into the cache. In contrast, our approach works on the traditional loop tiling.

Search-based techniques for finding tile sizes (and unroll factors) have received much attention in performance optimization [55,56,57,58,52]. The ATLAS system employs extensive empirical tuning to find the best tile sizes for different problem sizes in the BLAS library; tuning is done once at installation. Unfortunately such an approach is not suited for general tiled codes, as the search process is tuned for dense linear algebra codes only. Only square tile sizes are considered, which significantly hampers the performance of a variety of codes (such as stencils for instance) that require rectangular tiles for best performance. Furthermore, ATLAS currently includes a simplistic model where tile sizes are searched as to not exceed the square root of the L1 cache size. Our analytical bounds offer

a significantly higher accuracy, capturing both intra- and inter-tile reuse at various cache level.

Yotov et al. [98] have discussed the benefits of using model-driven approaches over empirical techniques for BLAS libraries, which closely relates to our approach. However, their approach proposed a single value of tile size as compared to our approach which creates DL-ML (lower-upper) bounds for tile size.

Kisuki et al. [52] have used different techniques such as genetic algorithms and simulated annealing to manage the size of the search space. Tiwari et al. [58] note: “a key challenge that faces auto-tuners, especially as we expand the scope of their capabilities, involves scalable search among alternative implementations.” The Active Harmony project [57, 58] uses several different algorithms to reduce the size of the search space such as the Nelder-Mead simplex algorithm. In contrast to these approaches, we use a pair of analytical models—a conservative model that overestimates the number of cache lines by ignoring lifetimes and an aggressive model that underestimates the number of cache lines—each leading to different sets of tile sizes, which are used to bound the search space. With our technique, any of the search algorithms [57, 58, 52] can be used to further reduce the search time.

6.2 Data Layout Optimization

Past research has proposed various data layout optimization techniques [12, 13, 14, 15]. Here, we present a brief survey of past work, focusing on aspects that are most closely

related to our work.

Zhang et al. [99] introduced a data layout framework that targets on-chip cache locality, specifically reducing shared cache conflicts while observing data patterns across threads. Using polyhedral analysis, their framework rearranges data layout tiles to reduce on-chip shared cache conflicts. However, their optimization currently works with single arrays. In contrast, our approach works on merging multiple arrays and operates at the element level rather than tiles.

Henretty et al. [100] presented a data layout framework to optimize stencil operations on short-SIMD architectures. Their work specifically targets stream alignment conflicts on vector registers and uses a dimension transposition method (non-linear data layout optimization) to mitigate the conflicts. In comparison, our approach works for more general applications, not just stencil benchmarks. Also, our work did not specifically address the impact of data layout on vectorization.

Ding and Kennedy [15] introduced a data-regrouping algorithm, which has similarities to our work on automatic selection of data layouts. Their compiler analysis merges multi-dimensional arrays based on a profitability cache analysis. Dynamic regrouping was also provided for layout optimization at runtime. Experimental results show significant improvement in cache and TLB hierarchy. However, their results were all obtained on uniprocessor systems and it is unclear how their approach works in the presence of data aliasing.

Raman et al. [101] used data layout transformations to reduce false sharing and im-

prove spatial locality in multi-threaded applications. They use an affinity based graph approach (similar to our approach) to select candidates. Inter-procedural aliasing issues arising due to pointers is not addressed in this work. Our work is intended to explore data layout transformations more broadly, not just for false sharing and spatial locality. Using polyhedral layout optimization, Lu et al. [102] developed a data layout optimization for future NUCA CMP architectures. Their work reduces shared cache conflict on such architectures. Simulation results show significant reductions in remote accesses. Finally, a number of papers, [103,104,105,106] have explored the integration of loop and data layout transformations.

Kremer [107] explored automatic data layout selection for distributed machines. He mapped the data layout problem into a 0-1 integer programming problem to find an optimal layout. Using an automatic framework, his tool explored choices of layouts for languages such as Fortran D or High Performance Fortran (HPF). This work explored possibilities of changing layouts across different phases (or loop partitions) in the same program. In contrast, our work explores data layout choices for a single node as compared to distributed machines and does not account for communication across the nodes. Our framework is based on choosing a global data layout for the entire program as opposed to region based layouts, due to expensive time spent in retransformation of a layout. Kremer's work explored multiple dimensions for a single array. However, in our current framework, we are limited with single dimensional arrays, including multi-dimensional arrays which have been transformed to single dimension.

To the best of our knowledge, our work is the first to support both user-specified and automatic AoS and SoA data layout transformations, while allowing the user to provide a data layout specification file. Our results (Section 4.2) on the LULESH mini-application demonstrates the importance of data layout transformations on modern multicore processors.

6.3 Distribution Function Selection

Fortran D Language [108, 109] introduced data distribution function across nodes. Block, Cyclic and Block-Cyclic were the three distributions primarily focused in this language. This functionality allowed programmer to choose an appropriate distribution function across different arrays in their program. Irregular distributions were handled by a separate distribution array, often named MAP. However, limited support was present for user-defined multi-dimensional functions. Computation or task partitioning was automatically derived using owner-computes rule from data distribution. Balasundaram et al. [110] developed a static performance estimator tool for predicting distributions across machines. This tool worked by training a performance model on a variety of kernel and then mapping source program to one of their training sets in order to select an efficient distribution. Using this approach, they predicted efficient distributions for REDBLACK benchmark.

In PARADIGM Compiler Framework, Manish Gupta et al. [111] developed an automatic distribution for arrays using owner computes rule. Their approach used cost based estimation for computation and communication to select block-cyclic distribution with vary-

ing block sizes.

High Performance Fortran (HPF) was a successor to the Fortran D language [112]. HPF provided further extensions to data distribution such as dynamic realignment and redistribution at runtime. In addition, it provided special directives (like SHADOW) to improve stencil computations on distributed nodes. The dHPF compiler and runtime system permitted dynamic selection of the dimensionality of a partitioning (2D-KD), the number of cuts per data dimension, and the mapping of tiles for multipartitionings based on data sizes and processor counts [113, 114]. This system employed computation partitioning framework to select optimal number of cuts to balance communication and computation across different nodes. The dHPF compiler automatically inserted appropriate MPI communication calls as compared to user specified communication calls.

Chapel programming language provides a more flexible and general framework for user specified data distribution with the standard block-cyclic distributions [115, 116].

In a separate but related domain, polyhedral methods map tiled code onto distributed node [117, 118]. Uday et al. [118] use polyhedral framework to automatically generate communication calls using MPI for affine loops, which minimizes communication and also satisfies data dependencies.

In contrast to the above work, our work is based on the Intel CnC model, which provides flexibility in user specified distribution function and also supports a wider set of programs as compared to affine-based models. In addition, our approach automatically selects a distribution function by using linear regression model. To the best of our knowledge, our

work is the first one to support dynamic graph analysis using regression model of CnC computations to automatically select an efficient distribution function.

Chapter 7

Conclusions and Future Work

Due to the ever increasing computing power capabilities, a plethora of processor architectures have emerged in modern times. One of the key challenges is extracting maximum performance from these emerging architectures. Programmers employ various optimization techniques to increase the application performance on processor architectures. However, varying architectures often require different optimizations for performance. Optimizations that take into account different architectural parameters help reduce this problem and enables *portable* performance.

In this dissertation, we have developed three approaches for portable locality optimizations: tile size selection, data layout optimization and distribution function selection. Our methods help improve performance of single node and multi-node environment. The first approach, tile size selection, introduces a novel model-driven empirical search using a new aggressive model named ML and conservative model named DL. Using these models as upper and lower bounds for tile size selection, our results demonstrate reduction in overall search space from $44 \times -11,879 \times$ across different architectures. These models help the programmer select efficient tile sizes as compared to exploring a larger search space across varying architectures. Our results show that for tile sizes that fall in-between the DL-ML model were 95% close to performance of optimal tile sizes.

Our second approach, data layout optimization, addresses the problem of selecting efficient data layouts across different platforms. We present a source-to-source framework for data layout optimization. Using our framework, programmers can specify a manual data layout or use our automated algorithm to select an efficient layout based on a given architecture. This approach avoids recurring changes to the source program for portable performance. Experimentation carried out using this method show performance improvements upto 20×.

Distribution function selection presents a new method to automatically select a distribution function based on a linear regression model as part of our third approach. While tuning applications in distributed environment, programmers have difficulty in selecting distribution functions and appropriate parameters for them. We present a linear regression model which uses sample runs for an application, learns the model and automatically selects a distribution function choice for a programmer. Our results show distribution function selection choices to be about 16% difference when compared to the optimal choice.

With these approaches, we show that efficient performance can be achieved from locality transformation. These transformations present computation and data optimizations for varying environments to enable minimal programmer effort. We believe that these approaches provide a sound foundation for future locality transformations and holds promising future directions.

Future Work

This dissertation opens new directions for future research. Some of those are listed below. However, this list is just a representation and directions can go much beyond these topics.

1. For work on tile size selection, the next step would be to develop auto-tuning learning models which would select tile sizes from the broader search space. These auto-tuning models would use dynamic profiles of the program to collect different metrics such as hardware performance counters to select an efficient tile size.
2. Our data layout optimization work transforms arrays into array of structs(AoS). Modern architectures such as Intel Xeon Phi, employ larger vector processing unit for performance efficiency. Extension to the data layout approach such as using array of structs of array (AoSoA), which benefits locality and vectorization appears to be a promising future approach.
3. From tile size selection and data layout optimization, we observe that locality optimization plays a key role in performance efficiency of an application. However, another important concern is energy optimization for application. Future research directions could study the impact of these transformation on energy optimization.
4. In the distribution function selection problem, we have focused on task distribution. In our work, we have used the default Intel CnC model, which places data where it is computed (owner-computes rule). Another future direction could be to see the

impact of separate task and data distribution.

5. As the number of cores and nodes increase in a distributed environment, leading to various constraints on the systems, fault tolerance will play a key role in applications. Faults can occur in different hardware components such as disk, memory and processor. Making applications resilient to these faults can be explored in the future along with performance efficiency.

Bibliography

- [1] J. Shirako, K. Sharma, N. Fauzia, L.-N. Pouchet, J. Ramanujam, P. Sadayappan, and V. Sarkar, “Analytical Bounds for Optimal Tile Size Selection,” in *Proceedings of the 21st International Conference on Compiler Construction, CC’12*, (Berlin, Heidelberg), pp. 101–121, Springer-Verlag, 2012.
- [2] K. Sharma, I. Karlin, J. Keasler, J. R. McGraw, and V. Sarkar, “User-specified and automatic data layout selection for portable performance,” Tech. Rep. TR13-03, Rice University, Houston, Texas, USA, April 2013. http://compsci.rice.edu/TR/TR_Download.cfm?SDID=307.
- [3] “The Platform-Aware Compilation Environment - Design Document,” September 2010. <http://pace.rice.edu/uploadedFiles/Publications/PACEDesignDocument.pdf>.
- [4] F. Irigoin and R. Triolet, “Supernode partitioning,” in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’88*, (New York, NY, USA), pp. 319–329, ACM, 1988.
- [5] M. Wolfe, “More iteration space tiling,” in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, Supercomputing ’89*, (New York, NY, USA), pp. 655–

- 664, ACM, 1989.
- [6] R. Schreiber and J. Dongarra, “Automatic blocking of nested loops,” Tech. Report 90.38, RIACS, NASA Ames Research Center, 1990.
- [7] J. Ramanujam and P. Sadayappan, “Tiling multidimensional iteration spaces for multicomputers,” *JPDC*, vol. 16, no. 2, pp. 108–230, 1992.
- [8] P. Boulet, A. Darte, T. Risset, and Y. Robert, “(Pen)-ultimate tiling?,” *Integration, the VLSI Journal*, vol. 17, no. 1, pp. 33–51, 1994.
- [9] J. Xue, *Loop tiling for parallelism*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.
- [10] M. Luersen, R. L. Riche, and F. Guyon, “A constrained, globalized, and bounded Nelder-Mead method for engineering optimization,” *Structural and Multidisciplinary Optimization*, vol. 27, no. 1-2, pp. 43–54, 2004.
- [11] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. Hollingsworth, “Scalable autotuning framework for compiler optimization,” in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, IPDPS '09*, May 2009.
- [12] T. M. Chilimbi, M. D. Hill, and J. R. Larus, “Cache-conscious structure layout,” in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, PLDI '99*, (New York, NY, USA), pp. 1–12, ACM, 1999.

- [13] B. Calder, C. Krintz, S. John, and T. Austin, “Cache-conscious data placement,” in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS VIII*, (New York, NY, USA), pp. 139–149, ACM, 1998.
- [14] T. M. Chilimbi and R. Shaham, “Cache-conscious coallocation of hot data streams,” in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '06*, (New York, NY, USA), pp. 252–262, ACM, 2006.
- [15] C. Ding and K. Kennedy, “Inter-array data regrouping,” in *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing, LCPC '99*, (London, UK, UK), pp. 149–163, Springer-Verlag, 2000.
- [16] “Intel CnC.” <http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc>.
- [17] M. Burke, K. Knobe, R. Newton, and V. Sarkar, “The Concurrent Collections Programming Model,” Tech. Rep. TR 10-12, Rice University, Houston, TX, USA, December 2010. http://compsci.rice.edu/TR/TR_Download.cfm?SDID=285.
- [18] A. J. Smith, “Cache memories,” *ACM Comput. Surv.*, vol. 14, pp. 473–530, Sept. 1982.
- [19] O. P. Agrawal and A. V. Pohm, “Cache Memory Systems for Multiprocessor Architecture,” in *Proceedings of the June 13-16, 1977, National Computer Conference*,

AFIPS '77, (New York, NY, USA), pp. 955–964, ACM, 1977.

- [20] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 3 ed., 2003.
- [21] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, “Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System,” in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques, PACT '09*, (Washington, DC, USA), pp. 261–270, IEEE Computer Society, 2009.
- [22] N. P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” in *Proceedings of the 17th Annual International Symposium on Computer Architecture, ISCA '90*, (New York, NY, USA), pp. 364–373, ACM, 1990.
- [23] B. Sinharoy, R. Kalla, W. Starke, H. Le, R. Cargnoni, J. Van Norstrand, B. Ronchetti, J. Stuecheli, J. Leenstra, G. Guthrie, *et al.*, “IBM POWER7 multicore server processor,” *IBM Journal of Research and Development*, vol. 55, no. 3, pp. 1–1, 2011.
- [24] “Intel Xeon Processor 5500 Series Datasheet, Vol. 1.” <http://www.intel.com/content/www/us/en/processors/xeon/xeon-5500-vol-1-datasheet.html>.
- [25] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, “Architecture of the IBM System/360,” *IBM J. Res. Dev.*, vol. 8, pp. 87–101, Apr. 1964.

- [26] K. Goto, “High-performance implementation of the level-3 blas,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 35, July 2008.
- [27] D. Levinthal, “Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors.” https://software.intel.com/sites/default/files/m/0/8/8/performance_analysis_guide.pdf.
- [28] B. Hall, M. Anand, B. Buross, M. Cilimdžić, H. Hua, J. Liu, J. MacMillan, S. Madali, K. Madhusudanan, B. Mealey, *et al.*, *POWER7 and POWER7+ Optimization and Tuning Guide*. IBM Redbooks, 2013.
- [29] W. A. Wulf and S. A. McKee, “Hitting the memory wall: Implications of the obvious,” *SIGARCH Comput. Archit. News*, vol. 23, pp. 20–24, Mar. 1995.
- [30] M. E. Wolf and M. S. Lam, “A data locality optimizing algorithm,” in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, PLDI '91*, (New York, NY, USA), pp. 30–44, ACM, 1991.
- [31] K. S. McKinley, S. Carr, and C.-W. Tseng, “Improving Data Locality with Loop Transformations,” *ACM Transactions on Programming Languages and Systems*, vol. 18, pp. 423–453, July 1996.
- [32] S. Carr, K. S. McKinley, and C.-W. Tseng, “Compiler optimizations for improving data locality,” in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS VI*, (New York, NY, USA), pp. 252–262, ACM, 1994.

- [33] K. Kennedy and K. S. McKinley, “Loop Distribution with Arbitrary Control Flow,” *Supercomputing '90*, November 1990.
- [34] M. Giles, G. Mudalige, C. Bertolli, P. Kelly, E. Laszlo, and I. Reguly, “An analytical study of loop tiling for a large-scale unstructured mesh application,” in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pp. 477–482, Nov 2012.
- [35] G. Rivera and C.-W. Tseng, “Tiling optimizations for 3D scientific computations,” in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, pp. 32–32, Nov 2000.
- [36] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.
- [37] D. F. Bacon, J.-H. Chow, D.-c. R. Ju, K. Muthukumar, and V. Sarkar, “A compiler framework for restructuring data declarations to enhance cache and tlb effectiveness,” in *Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON '94*, pp. 3–, IBM Press, 1994.
- [38] P. Ranjan Panda, H. Nakamura, N. Dutt, and A. Nicolau, “A data alignment technique for improving cache performance,” in *Proceedings of the 1997 International Conference on Computer Design (ICCD '97)*, ICCD '97, pp. 587–592, Oct 1997.

- [39] MPI Forum, “Message Passing Interface (MPI) Forum Home Page.”
<http://www.mpi-forum.org/> (Dec. 2009).
- [40] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, “Open MPI: Goals, concept, and design of a next generation MPI implementation,” in *Proceedings, 11th European PVM/MPI Users’ Group Meeting*, (Budapest, Hungary), pp. 97–104, September 2004.
- [41] T. El-Ghazawi, W. W. Carlson, and J. M. Draper, “UPC Language Specification v1.1.1,” October 2003.
- [42] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan, “Parametric multi-level tiling of imperfectly nested loops,” in *International Conference on Supercomputing (ICS)*, 2009.
- [43] M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan, “Parameterized tiling revisited,” in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO ’10*, pp. 200–209, 2010.
- [44] L. Renganarayanan, D. Kim, S. Rajopadhye, and M. M. Strout, “Parameterized tiled loops for free,” in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’07*, (New York, NY, USA), pp. 405–414, ACM, 2007.

- [45] D. Kim, L. Renganarayanan, D. Rostron, S. Rajopadhye, and M. M. Strout, “Multi-level tiling: M for the price of one,” in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC '07*, (New York, NY, USA), pp. 51:1–51:12, ACM, 2007.
- [46] S. Coleman and K. McKinley, “Tile Size Selection Using Cache Organization and Data Layout,” in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, PLDI '95*, (New York, NY, USA), pp. 279–290, ACM, 1995.
- [47] C. Hsu and U. Kremer, “A quantitative analysis of tile size selection algorithms,” *J. Supercomput.*, vol. 27, no. 3, pp. 279–294, 2004.
- [48] J. Chame and S. Moon, “A tile selection algorithm for data locality and cache interference,” in *ICS '99: Proceedings of the 13th international conference on Supercomputing*, (New York, NY, USA), pp. 492–499, ACM Press, 1999.
- [49] K. Esseghir, “Improving data locality for caches,” Master’s thesis, Dept. of Computer Science, Rice University, Sep. 1993.
- [50] S. Ghosh, M. Martonosi, and S. Malik, “Cache miss equations: a compiler framework for analyzing and tuning memory behavior,” *ACM TOPLAS*, vol. 21, no. 4, pp. 703–746, 1999.
- [51] G. Rivera and C. Tseng, “Locality optimizations for multi-level caches,” in *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing*

(*CDROM*), 1999.

- [52] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle, "Combined selection of tile sizes and unroll factors using iterative compilation," *The Journal of Supercomputing*, vol. 24, no. 1, pp. 43–67, 2003.
- [53] V. Sarkar, "Automatic Selection of High Order Transformations in the IBM XL Fortran Compilers," *IBM J. Res. & Dev.*, vol. 41, May 1997.
- [54] V. Sarkar and N. Megiddo, "An analytical model for loop tiling and its solution," in *Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS '00*, (Washington, DC, USA), pp. 146–153, IEEE Computer Society, 2000.
- [55] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1–2, pp. 3–35, 2001.
- [56] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology," in *Proceedings of the 11th International Conference on Supercomputing, ICS '97*, (New York, NY, USA), pp. 340–347, ACM, 1997.
- [57] C. Tapus, I.-H. Chung, and J. K. Hollingsworth, "Active Harmony: towards automated performance tuning," in *Proceedings of the 2002 ACM/IEEE Conference on*

- Supercomputing*, SC '02, (Los Alamitos, CA, USA), pp. 1–11, IEEE Computer Society Press, 2002.
- [58] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. Hollingsworth, “Scalable autotuning framework for compiler optimization,” in *23rd IEEE International Parallel & Distributed Processing Symposium Rome, Italy, Italy*, May 2009.
- [59] J. Navarro, A. Juan, and T. Lang, “MOB Forms: A Class of Multilevel Block Algorithms for Dense Linear Algebra Operations,” in *Proc. ACM International Conference on Supercomputing*, 1994.
- [60] K. Datta, “Auto-tuning stencil codes for cache-based multicore platforms,” technical report, University of California, Berkeley, Dec. 2009.
- [61] K. Yotov, K. Pingali, and P. Stodghill, “Think globally, search locally,” in *International Conference on Supercomputing*, 2005.
- [62] C. Chen, J. Chame, and M. Hall, “Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy,” in *International Symposium on Code Generation and Optimization (CGO'05)*, 2005.
- [63] A. Qasem and K. Kennedy, “Model-guided empirical tuning of loop fusion,” *International Journal of High Performance Systems Architecture*, vol. 1, no. 3, pp. 183–198, 2008.
- [64] J. Ferrante, V. Sarkar, and W. Thrash, “On estimating and enhancing cache effective-

- ness,” in *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, (London, UK, UK), pp. 328–343, Springer-Verlag, 1992.
- [65] T. W. Barr, A. L. Cox, and S. Rixner, “Translation caching: skip, don’t walk (the page table),” in *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 48–59, ACM, 2010.
- [66] “Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3A: System Programming Guide Part 1.”
- [67] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, “Accelerating two-dimensional page walks for virtualized systems,” in *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 26–35, ACM, 2008.
- [68] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, “A practical automatic polyhedral program optimization system,” in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08*, 2008.
- [69] U. Banerjee, *Dependence Analysis*. Kluwer Academic Publishers, 1997.
- [70] M. Lam, E. Rothberg, and M. Wolf, “The cache performance and optimizations of blocked algorithms,” in *Proceedings of the Fourth International Conference on Ar-*

chitectural Support for Programming Languages and Operating Systems, ASPLOS IV, pp. 63–74, 1991.

- [71] “Resource Characterization in the PACE Project.” <http://www.pace.rice.edu/Content.aspx?id=41>.
- [72] J. A. Nelder and R. Mead, “A simplex method for function minimization,” *Computer Journal*, vol. 7, no. 4, pp. 308–313, 1965.
- [73] V. Tabatabaee, A. Tiwari, and J. Hollingsworth, “Parallel Parameter Tuning for Applications with Performance Variability,” in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC ’05, pp. 57–57, 2005.
- [74] J. Keasler, T. Jones, and D. Quinlan, “TALC: A Simple C Language Extension For Improved Performance and Code Maintainability,” in *9th LCI International Conference on High-Performance Clustered Computing*, April 2008.
- [75] “TALC Infrastructure.” <https://wci.llnl.gov/codes/talc/index.html>.
- [76] D. Quinlan, “ROSE: Compiler support for object-oriented frameworks,” Tech. Rep. UCRL-ID-136515, Lawrence Livermore National Laboratory, 1999.
- [77] “Weblink.” <https://github.com/rose-compiler/edg4x-rose/tree/master/projects/TALCDataLayout>.
- [78] “ASC Sequoia Benchmark Codes.” <https://asc.llnl.gov/sequoia/benchmarks/>.

- [79] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pp. 44–54, IEEE, 2009.
- [80] “Hydrodynamics Challenge Problem,” Tech. Rep. LLNL-TR-490254, LLNL, Livermore, CA, USA, July 2011. <https://codesign.llnl.gov/pdfs/spec-7.pdf>.
- [81] I. Karlin, J. McGraw, J. Keasler, and C. Still, “Tuning the LULESH Mini-app for Current and Future Hardware,” in *Nuclear Explosive Code Development Conference Proceedings (NECDC12)*, December 2012.
- [82] J. D. McCalpin, “Memory Bandwidth and Machine Balance in Current High Performance Computers,” *IEEE TCCA Newsletter*, pp. 19–25, Dec. 1995.
- [83] Intel, “Intel 64 and IA-32 Architectures Optimization Reference Manual,” Tech. Rep. 248966-026, April 2012.
- [84] I.-H. Chung, C. Kim, H.-F. Wen, and G. Cong, “Application data prefetching on the ibm blue gene/q supercomputer,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12*, (Los Alamitos, CA, USA), pp. 88:1–88:8, IEEE Computer Society Press, 2012.
- [85] AMD, “Software Optimization Guide for AMD Family 15h Processors,” Tech. Rep. 47414, January 2012.

- [86] “AMD64 Architecture Programmer’s Manual Volume 2: System Programming.”
- [87] Budimlić, Zoran and Burke, Michael and Cavé, Vincent and Knobe, Kathleen and Lowney, Geoff and Newton, Ryan and Palsberg, Jens and Peixotto, David and Sarkar, Vivek and Schlimbach, Frank and Tasirlar, Saĝnak, “Concurrent collections,” *Sci. Program.*, vol. 18, pp. 203–217, Aug. 2010.
- [88] F. Schlimbach, K. Knobe, and J. Brodman, “Concurrent collections.” unpublished.
- [89] D. Zheng and T. Chang, “Parallel cholesky method on {MIMD} with shared memory,” *Computers and Structures*, vol. 56, no. 1, pp. 25 – 38, 1995.
- [90] A. George, M. T. Heath, and J. Liu, “Parallel Cholesky factorization on a shared-memory multiprocessor,” *Linear Algebra and its Applications*, vol. 77, no. 0, pp. 165 – 187, 1986.
- [91] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, “Communication-optimal parallel and sequential Cholesky decomposition: Extended abstract,” in *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA ’09, (New York, NY, USA), pp. 245–252, ACM, 2009.
- [92] “Boost Spirit.” <http://boost-spirit.com/home/>.
- [93] F. Bodin, W. Jalby, D. Windheiser, and C. Eisenbeis, “A quantitative algorithm for data locality optimization,” in *In Code Generation-Concepts, Tools, Techniques*, pp. 119–145, 1992.

- [94] T. Yuki, L. Renganarayanan, S. Rajopadhye, C. Anderson, A. Eichenberger, and K. O'Brien, "Automatic creation of tile size selection models," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pp. 190–199, 2010.
- [95] M. Christ, J. Demmel, N. Knight, T. Scanlon, and K. Yelick, "Communication lower bounds and optimal algorithms for programs that reference arrays – Part 1," Tech. Rep. UCB/EECS-2013-61, Electrical Engineering and Computer Sciences, University of California at Berkeley, July 2013.
- [96] B. Grey, J. Demmel, O. Holtz, and O. Schwartz, "Minimizing Communication in Numerical Linear Algebra," Tech. Rep. UCB/EECS-2011-15, Electrical Engineering and Computer Sciences, University of California at Berkeley, Feb. 2011.
- [97] I. Kodukula, N. Ahmed, and K. Pingali, "Data-centric Multi-level Blocking," *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, Las Vegas, Nevada*, pp. 346–357, June 1997.
- [98] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. A. Padua, K. Pingali, P. Stodghill, and P. Wu, "A comparison of empirical and model-driven optimization.," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI '03*, pp. 63–76, 2003.
- [99] Y. Zhang, W. Ding, J. Liu, and M. Kandemir, "Optimizing data layouts for parallel computation on multicores," in *Proceedings of the 2011 International Conference on*

Parallel Architectures and Compilation Techniques, PACT '11, (Washington, DC, USA), pp. 143–154, IEEE Computer Society, 2011.

- [100] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, “Data layout transformation for stencil computations on short-vector simd architectures,” in *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, (Berlin, Heidelberg), pp. 225–245, Springer-Verlag, 2011.
- [101] E. Raman, R. Hundt, and S. Mannarswamy, “Structure layout optimization for multithreaded programs,” in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, (Washington, DC, USA), pp. 271–282, IEEE Computer Society, 2007.
- [102] Q. Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Y. Chen, H. Lin, and T.-f. Ngai, “Data layout transformation for enhancing data locality on nuca chip multiprocessors,” in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, (Washington, DC, USA), 2009.
- [103] C. Ding and K. Kennedy, “Improving effective bandwidth through compiler enhancement of global cache reuse,” in *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, IPDPS '01, (Washington, DC, USA), IEEE Computer Society, 2001.

- [104] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee, “A framework for interprocedural locality optimization using both loop and data layout transformations,” in *Proceedings of the 1999 International Conference on Parallel Processing, ICPP '99*, (Washington, DC, USA), pp. 95–, IEEE Computer Society, 1999.
- [105] M. Taylan Kandemir, “Improving whole-program locality using intra-procedural and inter-procedural transformations,” *J. Parallel Distrib. Comput.*, vol. 65, pp. 564–582, May 2005.
- [106] M. F. P. O’Boyle and P. M. W. Knijnenburg, “Efficient parallelization using combined loop and data transformations,” in *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques, PACT '99*, (Washington, DC, USA), pp. 283–, IEEE Computer Society, 1999.
- [107] U. Kremer, *Automatic Data Layout for Distributed Memory Machines*. PhD thesis, Rice University, October 1995.
- [108] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu, “Fortran D Language Specification,” tech. rep., 1990.
- [109] S. Hiranandani, K. Kennedy, and C.-W. Tseng, “Compiling Fortran D for MIMD Distributed-memory Machines,” *Commun. ACM*, vol. 35, pp. 66–80, Aug. 1992.
- [110] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, “A static performance estimator to guide data partitioning decisions,” in *Proceedings of the Third ACM SIG-*

PLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '91, (New York, NY, USA), pp. 213–223, ACM, 1991.

- [111] M. Gupta and P. Banerjee, “Paradigm: A compiler for automatic data distribution on multicomputers,” in *Proceedings of the 7th International Conference on Supercomputing*, ICS '93, (New York, NY, USA), pp. 87–96, ACM, 1993.
- [112] High Performance Fortran Forum, “High Performance Fortran Language Specification Version 2.0,” tech. rep., Rice University Houston, TX, Oct. 1996.
- [113] A. Darte, J. Mellor-Crummey, R. Fowler, and D. Chavarría-Miranda, “Generalized multipartitioning of multi-dimensional arrays for parallelizing line-sweep computations,” *J. Parallel Distrib. Comput.*, vol. 63, pp. 887–911, Sept. 2003.
- [114] V. Adve, G. Jin, J. Mellor-Crummey, and Q. Yi, “High Performance Fortran compilation techniques for parallelizing scientific codes,” in *Proceedings of Supercomputing '98*, pp. 1–23, 1998.
- [115] R. Diaconescu and H. Zima, “An approach to data distributions in Chapel,” *Int. J. High Perform. Comput. Appl.*, vol. 21, pp. 313–335, Aug. 2007.
- [116] B. L. Chamberlain, S. J. Deitz, D. Iten, and S.-E. Choi, “User-defined distributions and layouts in Chapel: Philosophy and framework,” in *Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism*, HotPar'10, (Berkeley, CA, USA), pp. 12–12, USENIX Association, 2010.

- [117] M. Classen and M. Griebel, “Automatic code generation for distributed memory architectures in the polytope model,” in *Parallel and Distributed Processing Symposium (IPDPS), 2006.*, p. 7, 2006.
- [118] U. Bondhugula, “Compiling affine loop nests for distributed-memory parallel architectures,” in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis, SC '13*, (New York, NY, USA), pp. 33:1–33:12, ACM, 2013.