RICE UNIVERSITY

Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse

by

Chen Ding

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Ken Kennedy
Ann and John Doerr Professor, Chair
Computer Science

Keith Cooper
Associate Professor
Computer Science

Danny C. Sorensen
Professor
Computational and Applied Mathematics

Alan Cox
Associate Professor
Computer Science

John Mellor-Crummey
Senior Faculty Fellow
Computer Science

Houston, Texas
January 14, 2000

Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse

Chen Ding

Abstract

While CPU speed has been improved by a factor of 6400 over the past twenty years, memory bandwidth has increased by a factor of only 139 during the same period. Consequently, on modern machines the limited data supply simply cannot keep a CPU busy, and applications often utilize only a few percent of peak CPU performance. The hardware solution, which provides layers of high-bandwidth data cache, is not effective for large and complex applications primarily for two reasons: far-separated data reuse and large-stride data access. The first repeats unnecessary transfer and the second communicates useless data. Both waste memory bandwidth.

This dissertation pursues a software remedy. It investigates the potential for compiler optimizations to alter program behavior and reduce its memory bandwidth consumption. To this end, this research has studied a two-step transformation strategy: first fuse computations on the same data and then group data used by the same computation. Existing techniques such as loop blocking can be viewed as an application of this strategy within a single loop nest. In order to carry out this strategy to its full extent, this research has developed a set of compiler transformations that perform computation fusion and data grouping over the whole program and during the entire execution. The major new techniques and their unique contributions are

Maximal loop fusion: an algorithm that achieves maximal fusion among all program statements and bounded reuse distance within a fused loop.

Inter-array data regrouping: the first to selectively group global data structures and to do so with guaranteed profitability and compile-time optimality.

Locality grouping and **dynamic packing**: the first set of compiler-inserted and compiler-optimized computation and data transformations at run time.

These optimizations have been implemented in a research compiler and evaluated on real-world applications on SGI Origin2000. The result shows that, on average, the new strategy eliminates 41% of memory loads in regular applications and 63% in irregular and dynamic programs. As a result, the overall execution time is shortened by 12% to 77%.

In addition to compiler optimizations, this research has developed a **performance model** and designed a **performance tool**. The former allows precise measurement of the memory bandwidth bottleneck; the latter enables effective user tuning and accurate performance prediction for large applications: neither goal was achieved before this thesis.

Acknowledgments

I wish to thank my advisor, Ken Kennedy, for his inspiration, technical direction and material support. Without him this dissertation would not be possible. I want to thank my other committee members, Keith Cooper, John Mellor-Crummey, Alan Cox, and Danny Sorensen, for their interest and help. Sarita Adve helped with my proposal. I also thank Ellen Butler for always reserving me a slot in Ken's busy schedule. The implementation of my work was based on the D System, an infrastructure project led by John Mellor-Curmmey (and in part by Vikram Adve before his leave). I heavily used the scalar compiler framework put together by Nat Macintosh. The D System also contains components from previous compilers built by generations of graduate students.

I am very fortunate to study in a small department with leading researchers working in a close environment and in the same wonderful building. The professors and students of other groups not only give superb teaching but also are always ready to help. I thank in particular the language, scalar compiler, architecture and system group. My work was also helped by Nathaniel Dean and William Cook of computational mathematics department. My writing was significantly improved by a seminar taught by Jan Hewitt. In addition, I thank Ron Goldman for his valuable lunch-time advice and my officemate Dejan Mircevski for helping me on everything I asked. The financial support for my study came from Rice University, DARPA, and Compaq Corporation.

I received my M.S. degree from Michigan Tech., where my former advisors Phil Sweany and Steve Carr helped me to build a solid foundation for my research career. I thank also other outside researchers for their help especially Kathryn Knobe, Kathryn McKinley, Wei Li, and Chau-Wen Tseng.

This dissertation is dedicated to my family: my dear wife Linlin, my parents Shengyao Ding and Ruizhe Liu, and my brother Rui, for the never-ending love, support, and encouragement. I always remember what my father told me: "There are mountains after mountains and sky outside sky."

Contents

Illustrations

Chapter 1

Introduction

"In science there is only Physics; all the rest is stamp collecting" – Ernest Rutherford (1871-1937)

1.0 Thesis

At the dawn of the 21st century, the computing world is witnessing two powerful but diverging trends of hardware and software. On the hardware side, single-chip microprocessors have become the dominant platform for most applications simply because of their tremendous computing power, which has increased by an astonishing 6400 times in the past twenty years. However, in sharp contrast to the rapid on-chip improvement is the much slower rate of growth for off-chip memory bandwidth, which has increased by merely 139 times over the same period of time. To close the memory gap, all modern machines provide high-bandwidth on-chip data caches in the hope that most data can be cached so that applications can largely avoid direct access to memory.

Although caches have been successful for programs with small data sets and simple access patterns, their effectiveness has become increasingly problematic as the software community has been relentlessly pushing into ever larger and more complex systems. Not only do today's programs employ a massive amount of data that is far too large to fit in cache, they also access memory in a complex and dynamically changing manner that leads to extremely poor utilization of the available cache resource. The problem of poor cache utilization is further compounded by the use of module- or component-based programming styles that fragment both computation and data that could be otherwise cached together. As a result of the poor cache utilization and consequently poor memory performance, many applications can achieve only a few percent of peak CPU performance on modern machines, leaving room for a potential improvement of an order of magnitude if only caches could be better utilized.

The purpose of this thesis is to bridge the diverging trends of software and hardware by developing a compiler strategy that automatically transforms programs to fully utilize machine cache. Specifically, this work demonstrates that

> *Global and run-time transformations can substantially improve the overall performance of large, dynamic applications on machines built from modern microprocessors; furthermore, these transformations can be automated and combined into a coherent compiler strategy.*

The rest of this chapter first explains the problem of memory bottleneck and the solution of cache reuse. Then it presents the overall compiler strategy for maximizing cache reuse and compares this strategy with previous work. The succeeding chapters will then flesh out the various components of the new compiler strategy.

1.1 Problem of Memory Performance

The problem of memory performance is rooted in the diverging trends of hardware and software, in particular in the growing mismatch between the insufficient memory bandwidth supplied by machines and the massive memory transfer demanded by applications. This section first defines a few key concepts of memory hierarchy. The main part then studies the fundamental balance between computation and data transfer on computing systems, formulates a performance model based on the concept of balance, and finally uses the model to identify the performance bottleneck on modern machines.

1.1.0 Definitions

All modern machines built from microprocessors have data transferred through several levels of storage. The closest to CPU is a set of registers, then one or more levels of cache, and finally the main memory. This layered memory organization is called *memory hierarchy.*

Memory bandwidth is the data bandwidth between CPU and main memory, that is, how much data is communicated between them in each second. The communication is two-way: data is fetched into CPU through *memory reads* and sent back to memory by *memory writebacks.* The memory bandwidth of a program is called *effective memory bandwidth*, which is the number of memory reads and writebacks a program performs in each second. Since CPU and main memory are on different computer chips, the effective memory bandwidth of a program is constrained by the physical memory bandwidth of a machine, which is the cross-chip or off-chip hardware bandwidth between CPU and main memory.

Cache is a data buffer between CPU and main memory. It serves memory requests for the buffered data without accessing main memory. Cache is organized as a collection of non-unit cache blocks or cache lines. If a data item is buffered in cache, the whole block of the adjacent data is also loaded into the same cache block. A memory reference is a *cache hit* if the requested data is in a cache block; otherwise it is a *cache miss*, and the data is loaded directly from memory.

A repeated memory reference to the same data is a *data reuse*. If the requested data item is in cache, the data access is a *cache reuse*. Cache reuse may happen directly when the same data is requested twice, in which case the reuse is called a *temporal cache reuse*. Cache reuse may happen indirectly when a fresh data request hits in cache because the requested data has been brought in by the block transfer of a formerly requested data item, in which case the reuse is called a *spatial cache reuse*. Since large cache blocks are more efficient for contiguous data access and less costly for cache coherence, the size of cache blocks on modern machines is fairly large, ranging from 32 bytes to 128 bytes. Large cache blocks make cache spatial reuse extremely important for good cache utilization.

In the literature, cache spatial reuse is often defined differently in that it includes the fuzzy property that cache blocks do not unnecessarily conflict with each other to cause premature eviction from cache. This dissertation uses cache spatial reuse to denote only the reuse within a cache block; the conflicts among different cache blocks are referred to as *cache interference*.

1.1.1 Conflicting Trends of Software and Hardware

Since the advent of microprocessors in late 1970s, the capacity gap between off-chip memory bandwidth and on-chip CPU power has been steadily widening. Historical figures on processor performance and off-chip bandwidth have shown that over the past twenty years, the average annual increase in CPU power is 55%, but the average improvement in off-chip data bandwidth is merely 28%[1]. In other words, as CPU

---

[1] Estimation based on the historical figures compiled by Burger et al [BGK96].

power increased by 6,400 times in the past, memory bandwidth increased by no more than 139 times.

To bridge the memory gap, all modern machines provide high-bandwidth on-chip data cache in the hope that most memory reads and writebacks can be served by cache without consuming the valuable memory bandwidth. Although machine cache has been successful for programs with small data sets and simple access patterns, its effectiveness has become increasingly problematic because of the following directions pursued by modern software:

- *Large data sets:* A major goal in computing is to model the physical world, from a galaxy to a DNA, from an airplane to a robot, and from molecular dynamics to electromagnetism. Since we desire as large scope and as high precision as possible, the demand for larger data representations is insatiable.
- *Dynamic computation:* Most real-world events are non-uniform and evolving, such as that of a car crash or a drug injection. Consequently, both their computation structure and data representation are irregular and dynamically changing. Even in simpler cases where data stays the same, the order of data access may still change radically in different parts of a program. For example, a physical model can be traversed first top-down and then inside out.
- *Modularized programming:* To manage the complexity of developing software systems with sophisticated capabilities, modern software development must practice modularization along with computation and data abstraction. A computing task is frequently divided into a hierarchy of sub-steps, and a complex object broken into many sub-components.

Since large programs perform computation in many phases and access data in many different places, accesses to the same data item are far separated in time, and these accesses are often non-contiguous with large strides. When the reuse of a data item is far separated by a large amount of other data access, the value may be evicted from cache before it is reused, causing unnecessary data transfer from memory. Large-stride accesses, on the other hand, waste cache capacity by causing useless data to be transferred to cache. Furthermore, low utilization of cache blocks leads to an under-utilized cache, effectively reducing its size and causing even more memory transfer. Moreover, the extensive use of function and data abstraction aggravates the problem by fragmenting computations and data that could be otherwise cached together.

On parallel machines such as high-end servers and supercomputers, the problem of excessive memory transfer is as serious as it is on uni-processor machines. In fact, the bandwidth problem may cause even worse consequences for such machines because memory bandwidth is shared by a potentially large number of processors and consequently is a more critical resource. A single memory module can become the point of contention and the bottleneck of the whole parallel system. Recently, cache-coherent shared-memory multiprocessors have become increasingly popular because of their ease of programming. On such machines, a cache block is the basis of cache coherence and consequently the unit of inter-processor communication. Therefore, low cache-block utilization wastes not only memory bandwidth but also network bandwidth.

In summary, the analysis of hardware and software trends has revealed an alarming tension between the excessive demand of memory transfer and the limited supply of memory bandwidth. The next section examines the effect of this mismatch on performance.

1.1.2 Memory Bandwidth Bottleneck

This section quantifies the memory bandwidth constraint by modeling and measuring the fundamental balance between computation and data transfer.

Balance between Computation and Data Transfer

To understand the supply and demand of memory bandwidth as well as other computer resources, it is necessary to go back to the basis of computing systems, which is the balance between computation and data transfer. This section first formulates a performance model based on the concept of balance and then uses the model to examine the performance bottleneck on current machines.

Both a program and a machine have balance. *Program balance* is the amount of data transfer (including both data reads and writes) that the program needs for each computation operation; *machine balance* is the amount of data transfer that the machine provides for each machine operation. Specifically, for a scientific program, the program balance is the average number of bytes that must be transferred per floating-point operation (*flop*) in the program; the machine balance is the number of bytes the machine can transfer per flop in its peak flop rate. On machines with multiple levels of cache memory, the balance includes the data transfer between all adjacent levels.

The table in Figure 1.1 compares program and machine balance. The upper half of the table lists the balance of six representative scientific applications[2], including four kernels—convolution, dmxpy, matrix multiply, FFT—and two application benchmarks—SP from the NAS benchmark suite and Sweep3D from DOE. For example, the first row shows that for each flop, *convolution* requires transferring 6.4 bytes between the level-one cache (L1) and registers, 5.1 bytes between L1 and the level-two cache (L2), and 5.2 bytes between L2 and memory. The last row gives the balance of SGI Origin2000[3], which shows that for each flop at its peak performance, the machine can transfer 4 bytes between registers and cache, 4 bytes between L1 and L2, but merely 0.8 bytes between cache and memory.

As the last column of the table shows, with the exception of *mm(-O3)*, all applications demand a substantially higher rate of memory transfer than that provided by Origin2000. The demands are between 2.7 to 8.4 bytes per flop, while the supply is only 0.8 byte per flop. The striking mismatch clearly confirms the fact that memory bandwidth is a serious performance bottleneck. In fact, memory bandwidth is the least sufficient resource because its mismatch is much larger than that of register and

Programs	Program/machine Balance		
	L1-Reg	L2-L1	Mem-L2
convolution	6.4	5.1	5.2
dmxpy	8.3	8.3	8.4
mm (-O2)	24.0	8.2	5.9
mm (-O3)	8.08	0.97	0.04
FFT	8.3	3.0	2.7
NAS/SP	10.8	6.4	4.9
Sweep3D	15.0	9.1	7.8
Origin2000	4	4	0.8

Figure 1.1 Comparison between program and machine balance

---

[2] Program balances are calculated by measuring the number of flops, register loads/stores and cache misses/writebacks through hardware counters on SGI Origin2000.

[3] The machine balance is calculated by taking the flop rate and register throughput from hardware specification and measuring memory bandwidth through STREAM[McC95] and cache bandwidth through CacheBench[ML98].

cache bandwidth, shown by the second and the third column in Figure 1.1. The next section will take a closer look at this memory bandwidth bottleneck.

The reason matrix multiply *mm (-O3)* requires very little memory transfer is that at the highest optimization level of -O3, the compiler performs advanced computation blocking, first developed by Carr and Kennedy[CK89]. The dramatic change of results from -O2 to -O3 is clear evidence that a compiler may significantly reduce the application's demand for memory bandwidth; nevertheless, the current compiler is not effective for all other programs. I will return to compiler issues in a moment and for the rest of this dissertation.

Memory Bandwidth Bottleneck

The precise ratios of the demand of data bandwidth to its supply can be calculated by dividing the program balances with the machine balance of Origin2000. The results are listed in Figure 1.2. They show the degree of mismatch for each application at each memory hierarchy level. The last column shows the largest gap: the programs require 3.4 to 10.5 times as much memory bandwidth as that provided by the machine, verifying that memory bandwidth is the most limited resource. The data bandwidth on the other two levels of memory hierarchy is also insufficient by factors between 1.3 to 6.0, but the problem is comparatively less serious.

The insufficient memory bandwidth compels applications into unavoidable low performance simply because data from memory cannot be delivered fast enough to keep CPU busy. For example, the Linpack kernel *dmxpy* has a ratio of 10.5, which means an average CPU utilization of no more than 1/10.5, or 9.5%. One may argue

Applications	Ratios of demand over supply		
	L1-Reg	L2-L1	Mem-L2
convolution	1.6	1.3	6.5
dmxpy	2.1	2.1	10.5
mmjki (-O2)	6.0	2.1	7.4
FFT	2.1	0.8	3.4
NAS/SP	2.7	1.6	6.1
Sweep3D	3.8	2.3	9.8

Figure 1.2 Ratios of bandwidth demand to its supply

that a kernel does not contain enough computation. However, the last two rows show a grim picture even for large applications: the average CPU utilization can be no more than 16% for *NAS/SP* and 10% for *Sweep3D*. In other words, over 80% of CPU capacity is left unused because of the memory bandwidth bottleneck.

The memory bandwidth bottleneck exists on other machines as well. To fully utilize a processor of comparable speed as MIPS R10K on Origin2000, a machine would need 3.4 to 10.5 times of the 300 MB/s memory bandwidth of Origin2000. Therefore, a machine must have 1.02 GB/s to 3.15GB/s of memory bandwidth, far exceeding the capacity of current machines such as those from HP and Intel. As CPU speed rapidly increases, future systems will have even worse balance and a more serious bottleneck because of the lack of memory bandwidth.

So far, the balance-based performance model has not considered the effect of the latency constraint and, in particular, the effect of memory latency. It is possible that memory access incurs such a high latency that even the limited memory bandwidth is scarcely used. To verify that this is not the case, an additional study was performed to measure the actual bandwidth consumption of a group of program kernels and a full benchmark application, as reported in [DK00]. It found that these applications consume most of the available memory bandwidth. Therefore, memory bandwidth is a more limiting factor to performance than is memory latency.

In conclusion, the empirical study has shown that for most applications, machine memory bandwidth is between one third and one tenth of that needed. As a result, over 80% of CPU power is left un-utilized by large applications, indicating a significant performance potential that may be realized if the applications can better utilize the limited memory bandwidth. The next section introduces the solution developed by this dissertation: improving effective memory bandwidth through global and dynamic cache reuse.

1.2 Solution through Cache Reuse

This section starts with the general strategy of cache reuse, illustrates its power in exploiting global and dynamic cache reuse, demonstrates the necessity for its compiler automation, and finally presents the overall compiler strategy that systematically applies this strategy to maximize cache performance.

1.2.1 Two-Step Strategy of Cache Reuse

Cache reuse can be maximized by the following two-step strategy.

- Step 1. *fuse all the computation on the same data*
- Step 2. *group all the data used by the same computation*

The first step, computation fusion, groups all the uses of the same data so that when a data item is loaded into cache, the program performs all computation on that data before moving it out. The second step, data grouping, gathers all data used by the same computation so that during the computation, all cache blocks are utilized to the greatest extent possible. Both temporal and spatial cache reuse are maximized as a result of these steps.

Both steps have an implicit pre-step of separation before fusion and regrouping. The first step breaks computations into the smallest units before fusion so that unrelated computations are separated. Similarly, the second step divides data into the smallest pieces before regrouping so that unrelated data parts are disjointed. Therefore, the two-step strategy can be viewed as having four steps if the separation steps are made explicit.

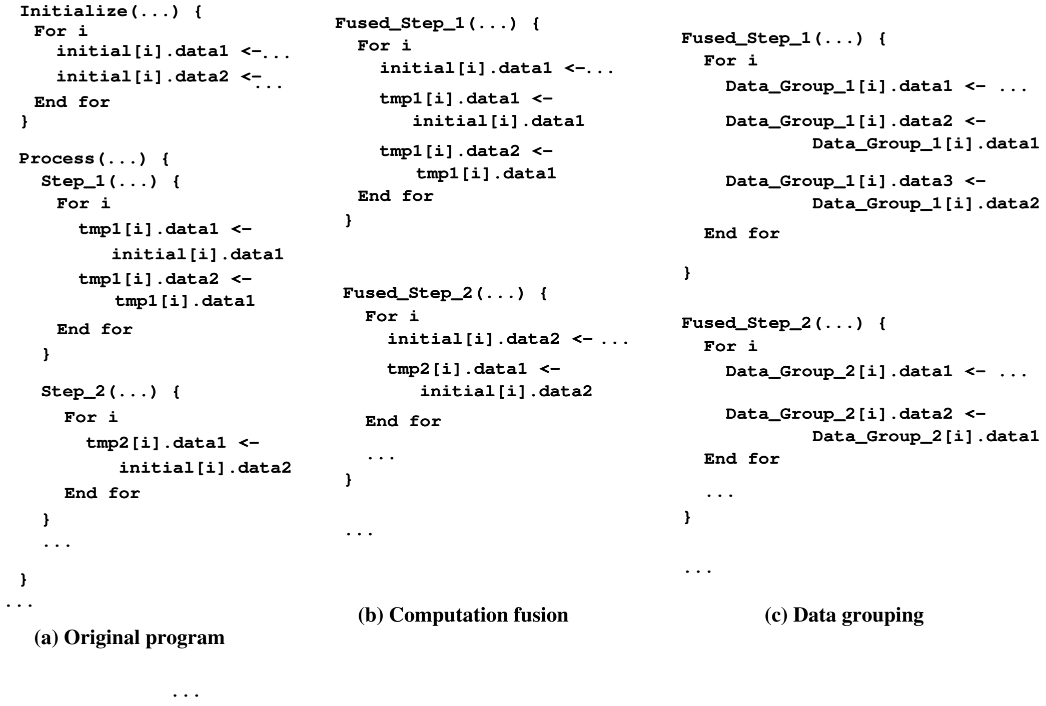
The strategy is a direct solution to the problems caused by far-separated reuse and large-stride access common in data-intensive programs. The fusion step minimizes the distance of data reuse, and the grouping step optimizes the stride of data access. As a result, computation fusion eliminates repeated memory transfer of the same data while data grouping fully utilizes each memory transfer. Together they minimize the total number of transferred cache blocks and therefore the total amount of memory bandwidth consumption.

The two steps of this strategy are inherently related: they are inseparable and they must proceed in order. The second step depends on the first because without fusion, data reuses remain far-separated and the repeated data access would miss in cache regardless of data grouping. On the other hand, the first step should be followed by the second because without data grouping, the cache and cache blocks may be polluted with useless data to the extent that only a few percent of cache is useful, and the effective memory bandwidth can be reduced by an order of magnitude. Therefore, neither step can work well without the other. This strategy and its benefits are especially evident when optimizing large and dynamic programs, as described in the next two sections and validated in the later chapters.

Global Cache Reuse

The strategy of cache reuse can be applied at the global level to improve data reuse across all program segments and in all data structures. Figure 1.3 illustrates global cache reuse. The example in (a) is a typical program written by a typical programmer. It starts with data initialization and then proceeds with several steps of computation. Although clear and simple logically, the program suffers from far-separated data reuse. For example, none of the input data is used until all other inputs are processed.

Computation fusion merges the computations on the same data, as shown in Figure 1.3(b). In the fused function *Fused_Step_1*, each data element is used immediately after its initialization, thus having a minimal reuse distance. Therefore, each element can be now buffered and reused with a fixed-size cache.



```
Initialize(...) {
 For i
   initial[i].data1 <-...
   initial[i].data2 <-...
 End for
}
Process(...) {
  Step_1(...) {
    For i
      tmp1[i].data1 <-
         initial[i].data1
      tmp1[i].data2 <-
         tmp1[i].data1
    End for
  }
  Step_2(...) {
    For i
      tmp2[i].data1 <-
         initial[i].data2
    End for
  }
  ...
}
...
```

(a) Original program

```
Fused_Step_1(...) {
  For i
    initial[i].data1 <-...
    tmp1[i].data1 <-
       initial[i].data1
    tmp1[i].data2 <-
       tmp1[i].data1
  End for
}

Fused_Step_2(...) {
  For i
    initial[i].data2 <- ...
    tmp2[i].data1 <-
       initial[i].data2
  End for
  ...
}
...
```

(b) Computation fusion

```
Fused_Step_1(...) {
  For i
    Data_Group_1[i].data1 <- ...
    Data_Group_1[i].data2 <-
            Data_Group_1[i].data1
    Data_Group_1[i].data3 <-
            Data_Group_1[i].data2
  End for
}

Fused_Step_2(...) {
  For i
    Data_Group_2[i].data1 <- ...
    Data_Group_2[i].data2 <-
            Data_Group_2[i].data1
  End for
  ...
}
...
```

(c) Data grouping

...

Figure 1.3 Example of global cache reuse

The fused program is not perfect because it makes scattered data access to different arrays. The second step, data grouping, gathers data used by the same computation into the same data array, as shown in Figure 1.3(c). After data grouping, not only are related data elements used together, they also locate together in physical memory. In combination, the fusion shortens temporal reuse between global computations, and the grouping increases spatial reuse among global data.

As shown by the example program, computation fusion and data grouping promise significant global benefit but also impose drastic changes to the whole program. Unlike localized techniques, a global transformation may move a piece of computation or data far away from its original place. New challenges immediately arise on maintaining correctness and estimating profitability. Interestingly, computation and data transformations follow different restrictions and cause different concerns. They raise different sets of questions.

Computation fusion is limited by data dependence. Given the widespread and complex dependences in real programs, how much fusion can a program have, or equivalently, how close can the uses of the same data be? Starting from that, how much can be achieved by a source-level transformation through a compiler? Since computation fusion may produce loops of a huge size, what is the overhead of fusion and how to eliminate or reduce that overhead? Chapter 2 will study computation fusion and address these challenging questions.

Unlike computation fusion, data grouping is not constrained by correctness because it does not violate any data dependence as long as a single storage is maintained for each program data. However, while fusion has no side effect on the unaltered program parts, data grouping uniformly affects every program segment that accesses the transformed data. In particular, data grouping in one place may not be beneficial for another place and may in fact be detrimental to overall performance. Therefore, the crucial problem of data grouping is evaluating its profitability: how to address the conflicting requirements of different program segments, and ultimately, how to find an optimal data layout for the whole program? Chapter 3 will study solutions to these problems.

Dynamic Cache Reuse

A large class of applications is dynamic, where some data structures and their access pattern remain unknown until run time and may change during the computation. An example is a car-crash simulation where the shape of the car remains unknown until the simulation starts, and the shape may change radically during the simulation.

To optimize a dynamic application, the strategy of cache reuse must be applied at run time after the computation and its data access are determined. Figure 1.4

illustrates dynamic data grouping. The example computation sequence traverses random elements of array f. The stride of access is large and varied. Data grouping first records the random data access and then gathers simultaneously used data into contiguous memory locations. If the data is accessed in the same or similar order multiple times, the overhead of grouping can be amortized effectively. With the transformed array shown in Figure 1.4, the dynamic access becomes more contiguous and obtains a better utilization of cache.

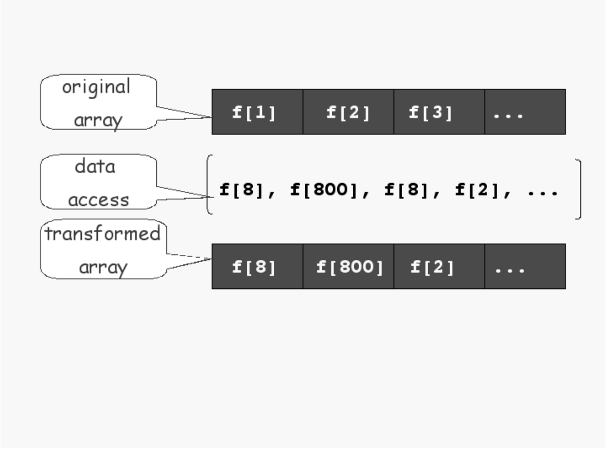


Figure 1.4 Example of dynamic cache reuse

Because of the unpredictable and dynamic nature of the computation and data, both analysis and transformation have to be performed at run time and probably be performed multiple times. Questions immediately arise on the feasibility, legality and profitability of such transformations. How to insert run-time analysis and code generation? What methods are cost-effective at run time? How to ensure their correctness, especially in the presence of repeated data layout changes? How much overhead do they incur, and can it be reduced through additional compiler optimizations? These questions will be addressed in Chapter 4.

1.2.2 The Need for Compiler Automation

Applying the strategy of cache reuse leads to radical program changes: computation fusion rewrites the whole program structure, and data grouping re-shuffles the entire data layout. In general, a program transformation may be carried out through three different agents: programmers, compilers, or hardware/operating systems. However, the global scope and extensive scale of computation fusion and data grouping suggest that an automatic compiler is the most viable approach. To demonstrate, Figure 1.5 lists the characteristics of all three options.

`approaches`	`advantages`	`disadvantages`
hardware or operating systems	√ precise run-time information	× very limited scope × run-time overhead of analysis and transformation
programmers	√ domain knowledge	× loss of function and data abstraction × inter-dependence between function and data
compilers	√ global scope √ off-line analysis and transformation	× imprecise program and machine information

Figure 1.5 Comparison among hardware/OS, programmers and compilers

Hardware and operating systems have precise knowledge of the operations being executed and the data being accessed. However, they cannot anticipate future: they can foresee at most a limited number of instructions down the executing path. Furthermore, because of the run-time overhead, they cannot afford extensive analysis and large-scale transformation, both of which are necessary for computation fusion and data grouping.

Programmers have domain knowledge of their applications. But manual computation fusion and data grouping render program abstraction and modularization impossible. Indeed, various functions must be mixed together if they access the same data; similarly, different data structures must be merged if they belong to the same computation. Furthermore, data layout now depends on computation structure. Whenever a memory access is added or deleted, the entire data layout may have to be reorganized. Therefore, if software development is to be scalable and maintainable, manual fusion and grouping should be mostly avoided.

Among all three approaches, only a compiler can afford the global scope and the extensive scale of computation fusion and data grouping. Given a source program, a compiler can analyze and transform the structure of both global computation and global data. The analysis and transformation are off-line without incurring any run-time overhead. A compiler, however, has its limitations. Its source-level analysis may not always accurate, and it cannot quantify the machine-dependent effect of a transformation. Despite its limitations, a compiler is currently the only viable choice to apply the strategy of cache reuse. If it succeeds, the benefit is enormous. The next section outlines such a compiler.

1.2.3 A Unified Compiler Strategy

This section presents a unified compiler strategy that maximizes memory hierarchy performance. It has four phases, as shown in Figure 1.6. The first two phases minimize overall memory transfer by maximizing cache reuse. The third phase schedules the remaining memory and cache access to tolerate its latency. The last phase engages user's help in identifying additional optimization opportunities that have been missed by automatic methods. The last column of Figure 1.6 lists the suitable techniques. Those developed by this dissertation are marked with a ⋆.

`main phases`	`sub-steps`	`suitable techniques` *(⋆ developed by this research)*
temporal reuse in cache and registers	global (multi-loop)	⋆ *maximal loop fusion*
	local (single loop)	unroll-and-jam, loop blocking, register allocation
	dynamic	⋆ *locality grouping*, space partitioning, curve ordering
cache-block reuse and cache utilization	inter-array spatial reuse	⋆ *inter-array data regrouping*
	intra-array spatial reuse	memory-order loop permutation, array reshaping, combined schemes
	dynamic spatial reuse	⋆ *dynamic data packing*
	cache non-interference	array padding, array copying, cache-conscious placement
latency tolerance	local (single loop)	data prefetching, instruction scheduling
user tuning	global (whole-program)	⋆ *model of machine & program balance* ⋆ *bandwidth-based performance tool*

Figure 1.6 The overall compiler strategy
for maximizing memory performance

The first phase converts data reuse into cache and register reuse. The primary method is computation fusion, which is first carried out at the global level across multiple loops, then at the local level within a single loop nest, and finally at run time for dynamic applications.

On an ideal machine with unit-size cache blocks, the first phase is sufficient for minimizing memory transfer. On a real machine, however, the second phase is needed to fully utilize non-unit cache blocks as well as memory pages. The first step of this phase exploits spatial reuse among global arrays. The succeeding steps improve spatial reuse within a single array both statically for regular programs and dynamically for dynamic applications. Finally, the last step adjusts the placement of large arrays to avoid the remaining cache interference.

After minimizing the amount of memory access by the first two phases, the third phase schedules the expensive memory and cache accesses so that their latency can be hidden as much as possible. The scheduling includes source-level data prefetching for high-latency memory access and assembly-level instruction scheduling for low-latency cache access. It should be noted that although latency tolerance is important, it does not help in ameliorating the memory bandwidth bottleneck as the previous phases do. In fact, data prefetching exacerbates the memory bandwidth problem because it causes additional memory transfer.

Compiler transformations, however, may still miss optimization opportunities or make imperfect transformations. When this happens, user tuning is necessary to achieve top performance. The last phase provides effective and efficient user tuning though a bandwidth-based performance tool. The tool can also provide accurate compile-time performance prediction, which is crucial for subsequent parallelization and run-time scheduling.

The global and dynamic techniques developed by this work play a vital role in the overall compiler strategy. The later chapters will describe these techniques and demonstrate their importance. The next section discusses existing local techniques and their limitations, as well as previous attempts at global and dynamic optimizations.

1.3 Related Work

This section surveys the techniques related to the overall compiler strategy, especially the previous work on global and dynamic transformations. Their limitations are first discussed individually and then summarized in the last section from three aspects: narrower purpose, lack of integrated transformation, and lack of compiler automation.

1.3.1 Complementary Techniques

Loop blocking and data prefetching are two widely used optimizations for memory hierarchy. They complement but cannot achieve the effect of global computation fusion and data grouping.

Loop Blocking

Loop blocking is a transformation that groups computations on sub-blocks of data that are small enough to fit in registers or in cache. A comprehensive study of blocking techniques can be found in Carr's dissertation[Car92]. The recent developments include the work by Kodukula et al[KAP97] and by Song and Li[SL99]. Since the new studies can implicitly optimize beyond a single loop nest, they will be discussed in the next section with the explicit work on loop fusion.

The primary limitation of loop blocking is its local scope: blocking is applied only to a single loop nest at a time. Consequently, it cannot exploit data reuse among disjoint loops. To overcome this limitation, we have to fuse multiple loops and determine how to interleave their iterations. This is precisely the process of loop fusion, which is discussed in the next section. Another limitation of blocking is that it cannot block computations and data that are unknown at compile time. Section 1.3.2 discusses related dynamic transformations.

Data Prefetching

Data prefetching is another widely studied technique. Unlike loop blocking or loop fusion, the goal of data prefetching is to tolerate or hide memory latency rather than to eliminate the memory access. Data prefetching identifies memory references that are cache misses and then dispatches them early enough in execution so that their latency can be overlapped with useful computation. Porterfield first developed software prefetching[Por89]. Mowry designed and evaluated a complete algorithm that later gained wide acceptance[Mow94].

Data prefetching, however, cannot hide memory latency imposed by the memory bandwidth bottleneck. Indeed, data prefetching does not reduce a single byte of memory transfer. On the contrary, it incurs additional memory transfer because it may prefetch the wrong data or prefetch too early or too late. Since actual memory latency is the reciprocal of the consumed memory bandwidth, data prefetching cannot completely hide memory latency unless the memory bandwidth bottleneck has been alleviated by other optimizations.

1.3.2 Global and Dynamic Optimizations

This section discusses previous work on global loop fusion, global data placement and dynamic optimizations.

Global Loop Fusion

Many researchers have studied loop fusion. Allen and Cocke first published the transformation[AC72]. The first significant role of fusion is to improve data reuse in a virtual memory system, studied by Abu-Sufah et al[ASKL81]. Wolfe gave a simple test for the legality of fusion[Wol82]. Two loops cannot be fused if they have fusion-preventing dependences, which are those forward dependences that are reversed after loop fusion. In the same work, Wolfe demonstrated through a few examples how loop fusion improves register reuse and reduces data storage on vector machines.

The first implementation of fusion in a compiler is by Allen[All83], who used loop fusion to improve register reuse in a legendary compiler that was later adopted by all vector supercomputers[AK87]. In its implementation, Allen required that fusible loops must have the same lower bound, upper bound and increment, no fusion-preventing dependence, and no true dependence on any intervening statements. Since the improvement by fusion is not as large as by other transformations such as loop interchange, Allen used fusion as a “cleanup” operation.

Loop fusion later took a prominent role in the work of Callahan, who used it to detect and construct coarse-grain parallelism[Cal87]. He gave a greedy fusion algorithm that runs in linear time to the number of loops and produces the minimal number of fused loops. The restriction for correctness is the same as in earlier studies, and the criterion for profitability is parallelism rather than cache reuse. So Callahan's method may fuse loops of no data sharing.

To enable more loop fusion, Porterfield introduced a transformation called *peel-and-jam*, which can fuse loops with fusion-preventing dependences by peeling off some iterations of the first loop and then applying fusion on the remaining parts[Por89]. While Porterfield considered only a pair of loops, Manjikian and Abdelrahman later extended peel-and-jam to find the minimal peeling factor for a group of fusible loops[MA97]. They evaluated their fusion scheme for parallel programs. Also enabled by peel-and-jam, Song and Li developed a new tiling method that blocks multiple loops within a time-step loop with the goal of improving cache reuse[SL99]. However, these methods are not a complete global strategy because they did not address the cases where not all loops in a program are fusible. In addition, peel-and-jam is a limited form of loop alignment because it can only shift the first loop up (or the second loop down), but not the reverse. So it does not always minimize the distance of data reuse in fused loops. Finally, peel-and-jam cannot fuse loops that have intervening statements that use the same data.

To find a solution for global loop fusion, a graph-partitioning formulation was studied independently both by Gao et al.[GOST92] and by Kennedy and McKinley[KM93]. Both their aims were to improve temporal reuse in registers, and they modeled the benefit of register reuse as weighted edges between a pair of loops. The goal was to partition all loops into legal fusible groups so that the inter-group edge weight (unrealized data reuse) is minimal. Kennedy and McKinley proved that the general fusion problem is NP-Complete. Both approaches used the heuristic that recursively applies min-cut algorithm to bi-partition the graph. Both avoided fusing loops with fusion-preventing dependences. However, a weighted-edge between two loops does not correctly model data sharing. Therefore, the partitioning method on normal graphs does not minimize the bandwidth consumption of the whole program. In another study of loop fusion, Darte considered the added complexity of loop shifting and proved that even loop fusion for single types (e.g. parallel loops) is strongly NP-complete in the presence of loop shifting[Dar99]. Recently, Kennedy developed a fast algorithm that always fuses along the heaviest edge[Ken99]. His algorithm allows accurate modeling of data sharing as well as the use of fusion enabling transformations. But none of these algorithms has been implemented or evaluated.

The first implementation for general fusion and its evaluation on non-trivial programs were accomplished by McKinley et al[MCT96]. They fused only loops with an equal number of iterations and with no fusion-preventing dependences. As a result, only 80 out of 1400, or 6% of tested loops were fused. The effect on full applications was mixed: fusion improved the hit rate for four out of 35 programs by 0.24% to 0.95%, but it also degraded performance of other three programs. Singhai and McKinley improved the fusion heuristic by considering the register pressure and by approximating graph partitioning with optimal tree partitioning[hSM97]. Since they fused only loops with no fusion-preventing dependences, the improvement to whole-program performance is modest except for two programs running on DEC Alpha. The potential of global data reuse is much larger, as demonstrated by a simulation study by McKinley and Temam[MT96]. They found that majority of program misses are inter-loop temporal reuses. Therefore, the important question remains open on the potential of global fusion, especially when aggressive fusion-enabling transformations are used.

To enable more aggressive loop fusion, some researchers have taken a radically different approach. Instead of blocking loops, Kodukula et al. tiled data and “shackled” computations on each data tile[KAP97]. Similarly, Pugh and Rosser sliced computations on each data element or data block[PR99]. Although effective for blocking single loops, data-oriented approaches are not yet practical as a global strategy for three reasons. First, without regular loop structures, it is not clear how to formulate and direct a global transformation. The shape of the transformed program is highly dependent on the choice of not only the shackled or sliced data but also of its starting loop. Furthermore, to maintain correctness, these methods need to compute all-to-all transitive dependences, whose complexity is cubic in the number of memory references in a program. Even when the dependence information is available, it is still not clear how to derive the best partitioning and ordering of the computations on different data elements, especially in the face of a large amount of unstructured computation. Finally, it is not clear how data-oriented transformations interact with traditional loop-based transformations, and how the side effect of fusion can be tackled. Kodukula et al. did not apply their work beyond a single loop nest[KAP97]. Pugh and Rosser tested *Swim* and *Tomcatv* and found mixed results. On SGI Octane, the first program was improved by 10% but the second “interacted poorly with the SGI compiler”[PR99].

The previous work on loop fusion did not combine it with data transformations with one exception. Manjikian and Abdelrahman, who applied padding to reduce cache conflicts[MA97]. Array padding at large data granularity is not a direct solution to cache utilization and has several important shortcomings compared to fine-grain data optimization, as discussed in the next section.

Global Data Placement

Once computation is optimized, data layout still needs careful arrangement because it affects the utilization within cache blocks and the interference among cache blocks.

Thabit studied the packing of scalars into cache blocks[Tha81]. He proved that finding the optimal packing for non-unit cache blocks is NP-complete.

The primary method for exploiting spatial reuse in arrays is to make data access contiguous. Instead of rearranging data, the early studies reordered loops so that the innermost loop traverses data contiguously within each array. Various loop permutation schemes were studied for perfect loop nests or loops that can be made perfect, including those by Abu-Sufah et al.[ASKL81], Gannon et al.[GJG88], Wolf and Lam[WL91], and Ferrante et al.[FST91]. McKinley et al. developed an effective heuristic that permutes loops into memory order for both perfect or non-perfect nested loops[MCT96]. Loop reordering, however, cannot always achieve contiguous data traversal because of data dependences. This observation led Cierniak and Li to combine data transformation with loop reordering[CL95], a technique that was subsequently expanded by Kandemir et al[KCRB98]. Regardless of the form of transformation, all these techniques are limited by their goal, which is to improve data reuse within a single array, or intra-array spatial reuse.

Data reuse within a single array is not adequate because not all data access to the same array can be made contiguous. One example is a dynamic application, where the data access within the same array is unpredictable at compile time, making it impossible to obtain contiguous memory access. Another example is a regular application, where the computation traverses high-dimensional data through different directions. Again, data access to a single array cannot always be made contiguous.

Data reuse among multiple arrays presents a promising alternative when data access can not be made contiguous. By combining multiple arrays and increasing the granularity of data access, the portion of useful data in each cache block can be significantly increased. In fact for large programs with many data arrays, inter-array reuse may fully utilize cache blocks without the need for contiguous data access. Inter-array data transformations, however, have not been attempted except for the work by Eggers and Jeremiassen[JE95]. They grouped all arrays accessed by a parallel thread to reduce false sharing among parallel processors. However, blindly grouping local data pollutes cache and cache blocks with useless data because not all local data objects are used at once. Besides the work on arrays, many researchers studied data placement optimizations for cache spatial reuse among pointer-based data structures. Seidl and Zorn clustered frequently referenced objects[SZ98], and Calder et al. reordered objects based on their temporal relations[CCJA98]. Chilimbi et al. clustered frequently used attributes within each object class[CDL99]. The basic approach shared by these methods is to place frequently used or closely referenced objects in close by memory locations. However, that two objects being either frequently accessed or for one time together accessed does not mean that they are always simultaneously accessed. Hence, their methods may place useless data into cache blocks and therefore degrade actual performance. In a large program where different data structures are used at different times, greedy grouping can seriously degrade cache-block reuse rather than improving it. Furthermore, these methods are static and therefore cannot fully optimize dynamic programs whose data access pattern changes during execution. For example, in a sparse-matrix code, the matrix may be iterated first by rows and then by columns. In scientific simulations, the computation order changes as the physical model evolves. In these cases, a fixed static data layout is not likely to perform well throughout the computation.

In addition to the reuse within the same cache block, attention needs to be paid to the interference among multiple cache blocks. A program can rearrange the location of whole arrays or array fragments in two ways: make them either well separated by padding, studied by Bailey[Bai92], or fully contiguous by copying, first used by Lam et al[LRW91]. Reducing cache interference, however, is not an approach as direct and effective as improving cache-block reuse. The best way to eliminate any cache interference is to place simultaneously used data into the same cache block, not by arranging them into multiple cache blocks. The large granularity used by packing precludes data reordering within the data object and across multiple data objects. Furthermore, padding cannot be applied to arrays of unknown size or machines with different cache parameters. It can reduce only cache interference but not the page-table working set. Moreover, both padding and copying carry a run-time cost, especially copying. Therefore, a compiler should first organize data within the same cache block and then use techniques such as data padding and copying to reduce cache interference if necessary.

Kremer developed a general formulation for finding the optimal data layout that is either static or dynamic for a program at the expense of being an NP-hard problem[Kre95]. He also showed that it is practical to use integer programming to find an optimal solution for normal programs. However, Kremer's formulation requires the estimation on the overhead and the benefit of a data transformation, which is not readily available to a compiler. He and others demonstrated that run-time communication and computation performance could be approximated through the use of training sets[BFKK91]. However, it is yet to be seen how well memory hierarchy performance can be predicted.

Dynamic Transformations

Researchers have long been studying dynamic applications such as molecular simulations. The best-known scheme is called inspector-executor, pioneered by Saltz and his colleagues[DUSH94]. At run time, the inspector analyzes the computation and produces an efficient parallelization scheme. Then the executor carries out the parallel execution.

Various specific schemes were also developed for optimizing cache performance. Saltz's group extended the inspector-executor model and used a reverse Cuthill Mcgee ordering to improve locality in a multi-grid computation[DMS$^+$92]. Another method, domain partitioning, has been used to block computation for cache by Tomko and Abraham[TA94]. Al-Furaih and Ranka examined graph-based clustering of irregular data for cache[AFR98]. Mellor-Crummey et al. employed space-filling curve ordering to block N-body type computations for multi-level memory hierarchy[MCWK99]. The above methods are powerful, but they incur a cost higher than linear to the number of data objects. Such cost becomes significant on large data sets and may not be cost effective for run-time readjustments. In addition, these transformations rely on the user knowledge. For example, the computation consists of interactions of either near by particles in a physical domain or neighboring nodes in an irregular graph.

Han and Tseng used a general scheme of grouping parallel computations accessing the same data object onto the same processor[HT98]. Although their transformation can be done in linear time and may be cost-effective for cache, they did not extend it to optimize cache performance. Mitchell et al. studied single non-affine memory references and used a more powerful method for partitioning, which is to sort irregular data access into "buckets"[MCF99]. Mitchell et al. discussed methods for automatically detecting opportunities for their optimization, but they did not show how to preserve the correctness by an automatic compiler.

A common limitation shared by all previous run-time techniques is the lack of general-purpose compiler automation. They targeted either a specific application domain or a very simplified computation model. The insufficient automation support limits the type of programs that can be handled and optimizations that can be used. As a result, large dynamic applications had to be transformed partially or wholly by hand. Since both the order of computation and the layout of data may be reorganized multiple times at run time, the code transformation process is extremely labor intensive and error prone. Even if a hand-optimized version is possible, it will be very difficult to maintain when new functions are added. Moreover, switching among and experimenting with different optimization schemes are even harder. Therefore, if run-time optimizations are to be practical and prevalent, they must be sufficiently automated.

1.3.3 Performance Model and Tool for Memory Hierarchy

Callahan et al. first used the concept *balance* to model whether register throughput can keep up with the CPU speed for scientific applications[CCK88]. However, they did not consider other levels of memory hierarchy.

In the past, monitoring memory hierarchy performance has to rely on machine simulators to gauge the exposed memory latency. Callahan et al. first used a compiler-based approach to analyze and visualize memory hierarchy performance with a memory simulator[CKP90] . Goldberg and Hennessy measured memory stall time by comparing actual running time with the simulation result of running the same program on a perfect memory[GH93]. Simulators, however, are inconvenient in practice because they are much slower than actual execution, and they are architecture-dependent. In addition, simulation-based approaches cannot be used for predicting memory hierarchy performance because it has to run the program before collecting its performance data.

Static or semi-static methods can be used to approximate run-time behavior and thus predict program performance. Bala et al. used training sets, which construct a database for the cost of various communication operations, to model communication performance in data-parallel programs [BFKK91]. They did not consider cache performance, although the same idea applies to cache. In another work, Clements and Quinn predicted cache performance by multiplying the number of cache-misses with memory latency[CQ93]. Their method is no longer accurate on modern machines, where memory transfers proceed in parallel with each other as well as with CPU computations. Moreover, they did not extend their work to support performance tuning.

Recently, researchers began to use bandwidth to measure machine memory performance. Examples are the STREAM benchmark by McCalpin [McC95] and CacheBench by Mucci and London [ML98]. However, neither of them explored the possibility of full-program tuning and performance prediction.

1.3.4 Summary of Limitations

No previous work has taken the goal of minimizing the total amount of data transfer between memory and CPU, nor has anyone explored the compiler strategy of global and dynamic computation fusion and data grouping. As a result, previous work shares the following three limitations:

Narrower purpose Previous techniques were not designed to solve the memory bandwidth problem, where single-loop based cache reuse is inadequate and latency tolerance is of no help. Therefore, global and dynamic cache reuse is the only software alternative to alleviate the memory bandwidth bottleneck. Failing to recognize this, existing techniques either do not address global and dynamic optimization or are not aimed at improving cache reuse. The former includes loop blocking; the latter, loop fusion and dynamic parallelization. Furthermore, none of previous studies addressed the problem of minimizing memory writebacks because they focused only on the latency of memory reads, not the bandwidth consumption of all memory access.

Lack of integrated transformation No previous work has successfully developed aggressive forms of computation fusion and data grouping, in part because these two steps have not been studied as a combined strategy. Without global data grouping, computation fusion may lead to extreme low cache utilization because the fused loop accesses too many dispersed data items. On the other hand, without aggressive fusion, data grouping may find little opportunity for combining global data structures since their accesses are separated in different parts of a program.

Lack of automation Because of the focus on memory latency, previous techniques are burdened with improving each memory reference individually, while neglecting the final goal of overall cache reuse of long computations on large data structures. These limitations lead to a preference of programmer-supplied or domain-specific transformations over compiler automation because of the possible compiler overhead. Unfortunately, manual or semi-manual techniques not only cannot master the scope and the scale of global computation fusion and data grouping, but they also lead to programming styles that are not maintainable and not portable. Furthermore, the focus on the latency of individual memory access makes performance modeling and debugging impractical, leading to ineffective user assistance for monitoring and tuning memory hierarchy performance.

1.4 Overview

To overcome the limitations of previous work, this dissertation has developed a new set of techniques that unleash the power of global and dynamic cache reuse. Chapter 2 describes global computation fusion, which exploits cache temporal reuse for the whole program. Chapter 3 presents inter-array data regrouping, which maximizes spatial reuse for the entire data. The dynamic transformations are described in Chapter 4, which include locality grouping for computation fusion and dynamic packing for data grouping. Chapter 5 complements these automatic techniques with a performance tool. The implementation and evaluation of all these techniques are described in Chapter 6. Finally, Chapter 7 summarizes the techniques having been developed and outlines their possible extensions.

> *"as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now that we have gigantic computers, programming has become an equally gigantic problem."* – Edsger W. Dijkstra, 1972

Chapter 2

Global Computation Fusion

"I hate quotations. Tell me what you know." – Ralph Waldo Emerson (1803-1882)

2.1 Introduction

As the first step to address the bandwidth limitation, this chapter explores the potential of global fusion in improving cache reuse over whole programs. The chapter investigates the following three problems. How is data reused in real programs? How beneficial is global fusion? And how much benefit can be realized by automatic transformations?

The chapter first defines *reuse distance*, a concept which precisely measures data reuse in a program before and after transformations. The chapter then studies two fusion transformations—one at the machine level and one at the program level. The machine-level model, *reuse-driven execution*, examines the potential of global fusion on an ideal machine, which always executes next the instructions that carry data reuse. More important is the source-level transformation, *maximal loop fusion*, which realizes the benefit of fusion on real machines. The main part of the chapter describes the algorithm of maximal loop fusion and shows that the new algorithm fuses loops whenever possible and achieves bounded reuse distance within a fused loop.

Although maximal fusion is the most aggressive in fusing global computations, it is not optimal. It does not minimize reuse distance within the fused loop, nor does it minimize the amount of data sharing among fused loops. The chapter formulates these problems and examines their complexity.

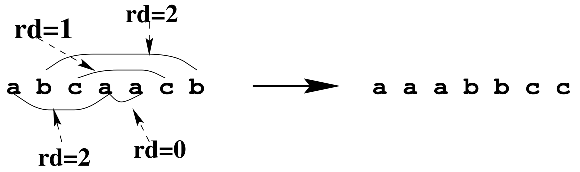
By bringing together all uses of the same data, global computation fusion shortens its live range. The localized data usage allows for aggressive storage transformations. The last part of the chapter describes two: *storage reduction* reduces the size of arrays, and *store elimination* removes memory writebacks to arrays.

2.2 Analysis of Data Reuse

This section first defines the concept of reuse distance and then explores the potential for minimizing reuse distances through reuse-driven execution.

2.2.1 Reuse Distance

In a sequential execution, the *reuse distance* of a data reference is the number of the *distinctive* data items appeared between this reference and the closest previous reference to the same data. The example in Figure 2.1(a) shows four data reuses and their reuse distance. On a perfect cache (fully associative with LRU replacement), a data reuse hits in cache if and only if its reuse distance is smaller than the cache size.



(a) Example sequence and its reuse distances

(b) Transformed data access sequence. All reuse distances are zero.

Figure 2.1 Example reuse distances

To avoid cache misses due to long reuse distances, a program can fuse computations on the same data. Figure 2.1(b) shows the computation sequence after fusion, where all reuse distances are reduced to zero. In general, the problem of finding minimal reuse distance can be reduced from the problem of weighted k-way cut[4]. The next section studies the use of heuristic-based fusion on real programs.

2.2.2 Reuse-Driven Execution

This section presents and evaluates *reuse-driven execution*, a machine-level strategy which fuses run-time instructions accessing the same data. In a sense, it is the inverse of Belady policy. While Belady evicts data that has the furthest reuse, reuse-driven execution executes the instruction that has the closest reuse. The insight gained in

---

[4] Section 2.4.1 studies this problem and demonstrates that a polynomial-time solution is unlikely because even the problem of unweighted 3-way cut is NP-complete.

this study will provide the motivation for the source-level transformation presented in the next section.

Given a program, its reuse-driven execution is constructed as follows. First, the source program is instrumented to collect the run-time trace of all source-level instructions as well as all their data access. The trace is re-run on an ideal parallel machine where an instruction is executed as soon as all its operands have been computed. The trace of an ideal execution gives the ordering of instructions and their minimal time difference. Finally, reuse-driven execution is carried out by the algorithm given in Figure 2.2. It is reuse-driven because it gives priority of execution to later instructions that reuse the data of the current instruction. It employs a FIFO queue to sequentialize the execution of instructions.

The effect of reuse-driven execution is shown in Figure 2.3 for a kernel program *ADI* and an application benchmark *NAS/SP* (Serial version 2.3); the former has 8 loops in 4 loop nests, and the latter has over 218 loops in 67 loop nests. In each figure, a point at (x, y) indicates that y thousands of memory references have a reuse distance between $[2^{(x-1)}, 2^x)$. The figure links discrete points into a curve to emphasize the elevated hills, where large portions of memory references reside. The important measure is not the length of a reuse distance, rather it is whether the length increases with the input size. If so, the data reuse will become a cache miss when data input is sufficiently large. We call those reuses whose reuse distance increases with the input size *evadable reuses*.

The upper two figures of Figure 2.3 show the reuse distances of *ADI* on two input sizes. The two curves in each figure show reuse distances of the original program and that of reuse-driven execution. In the original program, over 40% of memory references (25 thousand in the first and 99 thousand in the second) are evadable reuses. However, reuse-driven execution not only reduced the number of evadable reuses by 33% (from 40% to 27%), but also slowed the lengthening rate of the remaining evadable reuses.

A similar improvement is seen on *NAS/SP*, where reuse-driven execution reduced the number of evadable reuses by 63% and slowed the rate of lengthening of reuse distances.

We also tested two other programs—a *FFT* kernel and a full application, *DOE/Sweep3D*, shown in Figure 2.4. Reuse-driven execution did not improve *FFT* (where the number of evadable reuses was increased by 6%), but it reduced evadable reuses by 67% in *DOE/Sweep3D*. In addition, other heuristics of reuse-driven execu-

```
function Main
   for each instruction i in the ideal parallel execution order
       enqueue i to ReuseQueue
       while ReuseQueue is not empty
           dequeue instruction i from ReuseQueue
           if (i has not been executed)
              ForceExecute(i)
       end while
   end for
end Main

function ForceExecute(instruction j)
   while there exists un-executed instruction i that produces operands for j
       ForceExecute(i)
   end while
   execute j
   for each variable t used by j
       find the next instruction m that uses t
       enqueue m into ReuseQueue
   end for
end ForceExecute
```

Figure 2.2 Algorithm for reuse-driven execution

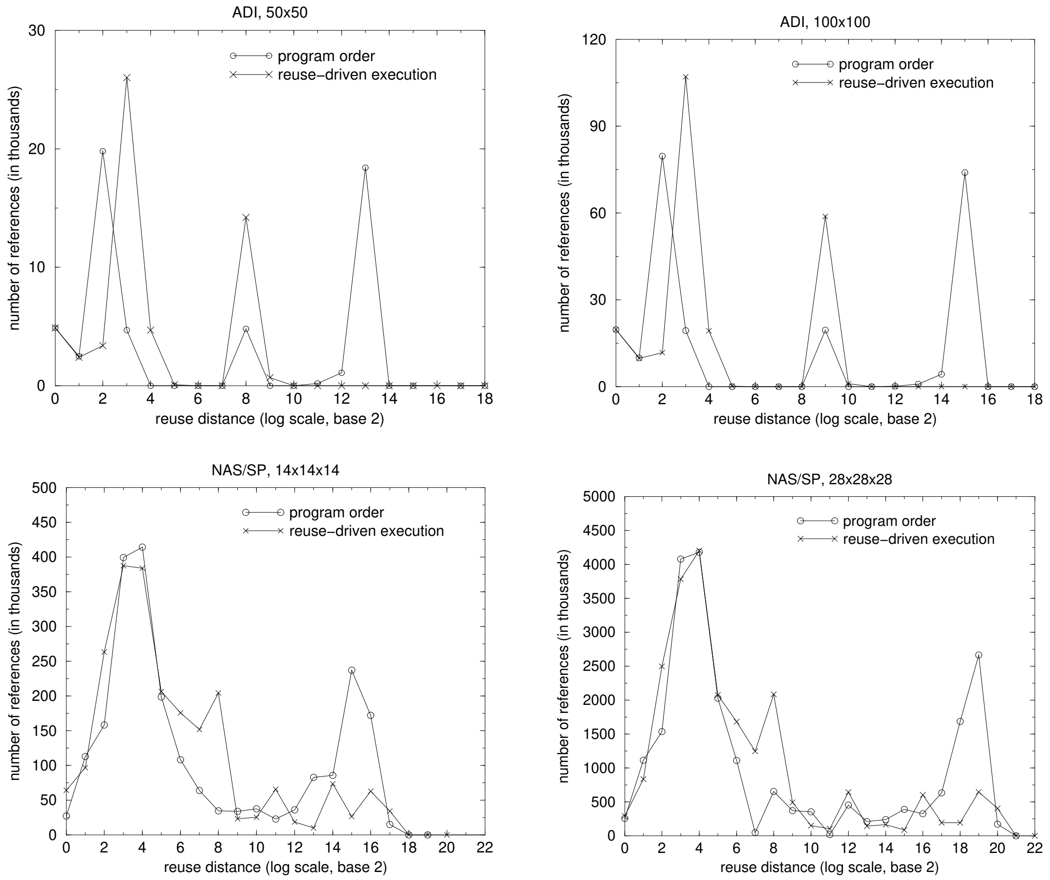


Figure 2.3 Effect of reuse-driven execution (I)

tion were also evaluated. For example, that of not executing the next reuse if it is too far away (in the ideal parallel execution order). But the result was not improved. The experiment with reuse-driven execution demonstrates the potential of fusion as a global strategy for reducing the number of evadable reuses in large applications with multiple loop nests. The next section studies aggressive loop fusion as a way to realize this benefit. The effect of loop fusion on reuse distances will be measured in Chapter 6.

2.3 An Algorithm for Maximal Loop Fusion

Since loops contain most data access and data reuse, loop fusion is obviously a promising solution for shortening reuse distances. The first half of this section presents an

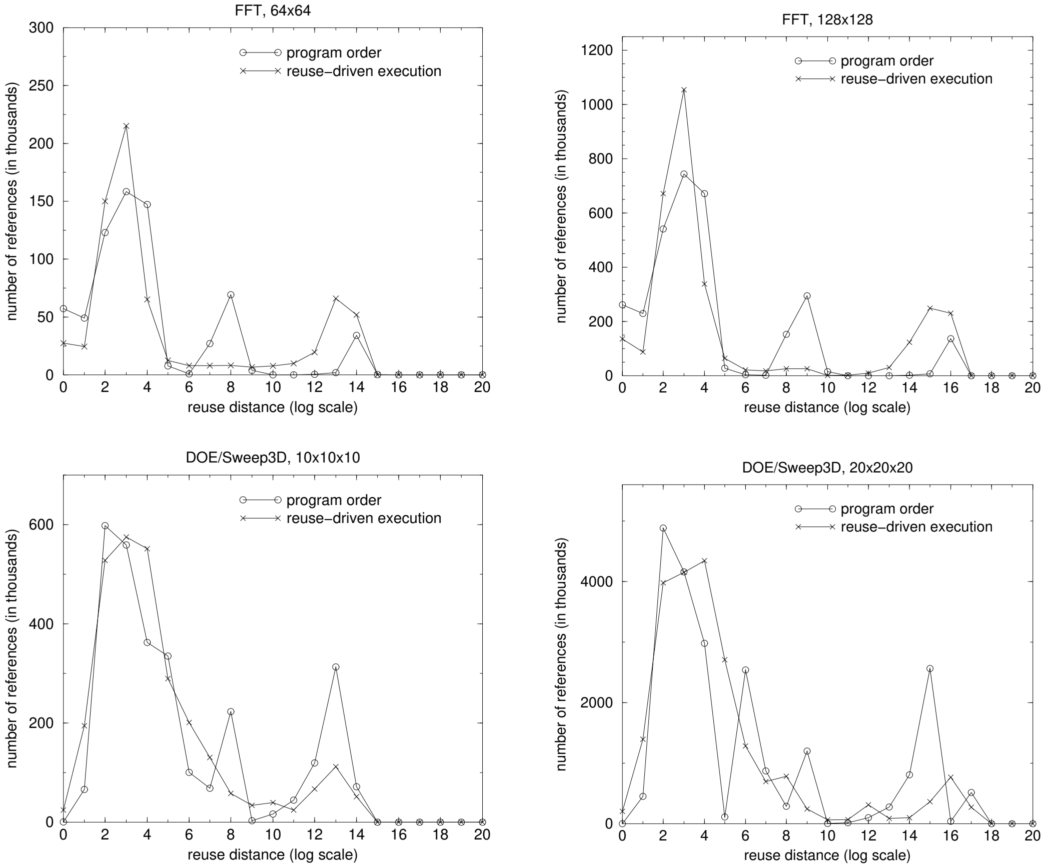


Figure 2.4 Effect of reuse-driven execution (II)

efficient algorithm that achieves maximal loop fusion and bounded reuse distance. The second half formulates the problem of optimal loop fusion and then studies its complexity. Although the following discussion assumes that a program is structured in loops and arrays, the formulation and solution to loop fusion apply to programs in other language structures such as recursive functions and object-based data.

The example program in Figure 2.5(a) has two loops sharing the access to array A. They cannot be fused directly because of the two intervening statements that also access part of A. To enable loop fusion, we need three supporting transformations. The first is *statement embedding*, which fuses the two non-loop statements into the first loop. It schedules `A[2]=0.0` in the second iteration, where $A[2]$ is last used. Similarly, it puts `A[1]=A[N]` in the last iteration, where $A[N]$ is last computed.

```
    for i=2, N
      A[i] = f(A[i-1])
    end for
s3  A[1] = A[N]
s4  A[2] = 0.0
    for i=3, N
      B[i] = g(A[i-2])
    end for
```

```
for i=2, N
   A[i] = f(A[i-1])
   if (i==3)
     A[2] = 0.0
   else if (i == N)
     A[1] = A[N]
   end if
  if (i>2 and i<N)
     B[i+1] = g(A[i-1])
   end if
end for
B[3] = g(A[1])
```

(a) fusion by statement embedding, loop alignment and loop splitting

```
for i=2,N
  A[i] = f(A[i-1])
end for
A[1] = A[N]
for i=2,N
  A[i] = f(A[i-1])
end for
```

(b) example of loops that cannot be fused

Figure 2.5 Examples of loop fusion

After statement embedding, two loops are still not directly fusible because the first iteration of the second loop depends on the last iteration of the first loop. The second transformation, *iteration reordering*, splits the second loop and peels off its first iteration so that the remaining iterations can be fused with the first loop.

When two loops are fused, the third transformation, *loop alignment*, ensures that their iterations are properly aligned. The second loop is shifted up by one iteration so that the reuse of $A[i-1]$ happens within the same iteration. Otherwise, the reuse would be one iteration apart, unnecessarily lengthening reuse distance. The fused program is shown in Figure 2.5(a), where array A is closely reused.

Loop fusion introduces instruction overhead to the fused program because of the inserted branch statements. Although this overhead was prohibitively high for previous-generation machines, today's fast processors on modern machines can easily offset this additional cost. Chapter 6 will measure the effect of fusion on real machines.

Although the supporting transformations enable loop fusion in this example, they do not always succeed. For example, the two loops in Figure 2.5(b) can never be fused because all iterations of the second loop depend on all iterations of the first loop. The dependences caused by the intervening statements make fusion impossible. Since the feasibility test of fusion has to consider the effect of non-loop statements, the cost can be too high if loop fusion is tested for every pair of loops. To avoid this cost, the following algorithm employs incremental fusion, which examines only the closest pair of data-sharing loops for fusion.

2.3.1 Single-Level Fusion

The following discussion of loop fusion makes several assumptions as listed in Figure 2.6. At the beginning, we consider only single-dimensional loops accessing single-dimensional arrays. Later we will use the same algorithm to fuse multi-dimensional loops level by level. The other restrictions in Figure 2.6 can also be relaxed, however, at the cost of a more complex fusion algorithm. For example, index expressions like $A(d * i + c)$ can be considered by projecting the sparse index set into a dense index set.

- a program is a list of loop and non-loop statements
- all loops are one-dimensional and so are all variables
- all data accesses are in one of the two forms: $A[i + t]$ and $A[t]$, where A is the variable name, i is the loop index, and t is a loop-invariant constant

Figure 2.6 Assumptions on the input program

The fusion algorithm is given in Figure 2.7, which incrementally fuses all data-sharing loops. For each statement $p[i]$, subroutine *GreedilyFuse* tries to fuse it upwards with the closest predecessor $p[j]$ that accesses the same data. If $p[i]$ is a statement, it can be embedded into $p[j]$. Otherwise, subroutine *FusibleTest* is called to test whether the two loops can be fused. If $p[i]$ is fused with $p[j]$, *GreedilyFuse* is recursively applied on $p[j]$ because it now accesses a larger set of data.

Subroutine *FusibleTest* determines whether two loops can be fused, and if so, what reordering is needed and what the minimal *alignment factor* is. An alignment factor of k means to shift the iterations of the second loop down by k iterations. The alignment factor can be negative, when the second loop is shifted up to bring together data reuses. For each data array, the subroutine determines the smallest alignment factor that both satisfies data dependence and has the closest data reuse. To avoid unnecessarily increasing the alignment factor, the algorithm does not allow positive alignment factors for read-read data reuse. The final alignment factor is the largest found among all arrays. The algorithm avoids repeated *FusibleTest* by remembering infusible loop pairs.

A fused loop is represented as a collection of loop and non-loop statements, where loops are aligned with each other, and non-loop statements are embedded in some iteration of the fused loop. The data footprint of a loop includes the access to all

```
SingleLevelFusion
  let p be the list of program statements, either loop or non-loop statements
  iterate p[i] from the first statement to the last in the program
     GreedilyFuse(p[i])
end SingleLevelFusion

Subroutine GreedilyFuse(p[i])
  search from p[i] to find the most recent predecessor p[j] sharing data with p[i]
  if p[j] does not exist, exit and return
  if (p[i] is not a loop)
     embed p[i] into p[j]
     make p[i] an empty statement
  else if (FusibleTest(p[i], p[j]) finds a constant alignment factor)
     if (no splitting is required)
        fuse p[i] into p[j] by aligning p[i] by the alignment factor
        make p[i] an empty statement
        GreedilyFuse(p[j])
     end if
     if (splitting is required)
        split p[i] and/or p[j] and fuse p[i] into p[j] by aligning p[i]
        make p[i] an empty statement
        GreedilyFuse(p[j])
        for each remaining pieces t' after splitting
            GreedilyFuse(t')
     end if
  end if
end FuseStatement

Subroutine FusibleTest(p[i], p[j])
  if (p[i] p[j]) has been marked as not fusible
     return false
  end if
  for each array accessed in both p[i] and p[j]
     find the smallest alignment factor that
         (1) satisfies data dependence, and
         (2) has the closest reuse
     apply iteration reordering if necessary and possible
  end for
  find the largest of all alignment factors
  if (the alignment factor is a bounded constant)
     return the alignment factor
  else
     mark (p[i] p[j]) as not fusible
     return false
  end if
end FusibleTest
```

Figure 2.7 Algorithm for one-level fusion arrays. For each array, the data access consists of loop-invariant array locations and loop-variant ranges such as $[i+c_1, i+c_2]$, where i is the loop index and c_1 and c_2 are loop-invariant constants. Data dependences and alignment factors are calculated by checking for non-empty intersections among footprints.

2.3.2 Properties

Maximal Fusion The three transformations achieve maximal fusion. Statement embedding can always fuse a non-loop statement into a loop. Loop alignment avoids conflicting data access between two loops by delaying the second loop by a sufficient factor. When a bounded alignment factor cannot be found, iteration reordering is used to extracts fusible iterations and arranges them in a fusible order. Examples of iteration reordering include loop splitting and loop reversal. By employing these three transformations, the algorithm in Figure 2.7 fuses two loops whenever (1) they share data and (2) their fusion is permitted by data dependence. Therefore, the algorithm achieves maximal fusion.

Bounded Reuse Distance The length of reuse distances are bounded after loop fusion, as proved in the following theorem. The restrictions listed in Figure 2.6 are implicitly assumed throughout this section.

> **Theorem 2.1** In a fused loop, if the effect of loop-invariant data accesses is excluded, the reuse distance of all other data accesses is not evadable, that is, the reuse distance of all loop-variant accesses does not increase when the input size grows.

Proof The reuse distance between two uses of the same data is bounded by the product of the number of iterations between the two uses and the amount of data accessed in each iteration. Next we examine the maximal value of these two terms.

The iteration difference between two data-sharing statements increases only because of loop alignment. Since the alignment factor between each pair of loops is a constant, the total iteration difference between any two fusible loops is at most $O(L)$, where L is the number of loops in the program.

Excluding loop-invariant accesses, a fused loop accesses a collection of loop-variant ranges. A loop has at most $O(A)$ such ranges, where A is the number of arrays in the program. In a given iteration, a loop-variant range includes a constant number of data elements (because of the restrictions made in Figure 2.6). Therefore, each iteration accesses at most $O(A)$ data elements.

Since two uses of the same data is at most $O(L)$ iterations apart with at most $O(A)$ elements in each iteration, the upper bound on the reuse distance is $O(A * L)$, which is independent of the sizes of arrays. □

The upper bound on reuse distance, $O(A{*}L)$, is tight because a worst-case example can be constructed as follows: the first loop is `B(i)=A(i+1)`, then are L loops of `B(i)=B(i+1)`, finally is `A(i)=B(i)`. Since the two accesses to $A(i)$ must be separated by L iterations, the reuse distance can be no less than L. Therefore, the fusion algorithm achieves the tightest asymptotic upper bound on reuse distances.

Fast Algorithm The following theorem gives the time complexity of the fusion algorithm in Figure 2.7. The cost is in fact smaller in practice because a restricted version of loop fusion suffices for all tested programs, as explained after the theorem.

> **Theorem 2.2** The time complexity for the algorithm is $O(V * V' * (T + A))$, where V is the number of program statements before fusion, V' is the number of fused loops after fusion, T is the cost of *FusibleTest*, and A is the number of data arrays in the program.

Proof The complexity of the fusion algorithm is the number of invocations of *GreedilyFuse* times its cost. *GreedilyFuse* is called for each program statements first and then for each new loop generated by fusion. In a program of V program statements, fusion generates at most V new loops (each successful fusion decreases the number of loops by one). Therefore, the number of invocations of *GreedilyFuse* is $O(V)$.

The cost of each *GreedilyFuse* includes the cost of (1) finding the most recent data-sharing loop, (2) fusing two statements, and (3) checking the fusibility by *FusibleTest*. The data-sharing loop can be found by a backward search through fused loops. The cost is $O(V' * A)$, where V' is the number of loops after fusion. Fusing two statements requires updating the data footprint information, the cost of which is $O(A)$. When examining a fusion candidate, *FusibleTest* is invoked at most once for each fused loop, so the number of invocations is $O(V * V')$, discounting the additional loops created by iteration reordering. Each invocation of *FusibleTest* takes $O(A)$ to check all arrays of a footprint. The cost of iteration reordering is assumed to be T. Hence, the total cost of *FusibleTest* is $O(V * V' * (A + T))$. The remaining cost of *GreedilyFuse* is $O(V * (V' * A + A))$. Therefore, the total cost of the algorithm is $O(V * V' * (A + T))$. $\square$

The implementation in Chapter 6 makes two simplifications. It assumes that all loop-invariant array accesses are on bordering elements, and it reorders iterations only by splitting at boundary loop iterations. As shown in the evaluation chapter, these two assumptions are sufficient to capture all possible fusion in the programs we tested. In the simplified algorithm, the cost of each *FusibleTest* is $O(A)$. Therefore, the time complexity of fusion is $O(V * V' * A)$. In a typical program where the number of fused loops and the number of arrays are orders of magnitude smaller than the number of program statements, the cost of simplified loop fusion is approximately linear to the size of the program.

2.3.3 Multi-level Fusion

The previous sections have assumed loops and arrays of a single dimension. For programs with multi-dimensional loops and arrays, the same fusion algorithm can be applied level by level as long as the ordering of loop and data dimensions is determined. Figure 2.8 gives the algorithm for multi-level fusion, which maximizes the overall degree of fusion by favoring loop fusion at outer loop levels.

While all data structures and loop levels are used to determine the correctness of fusion, only large data structures are considered in determining the profitability of fusion because the sole concern is the overall data reuse. In the following discussion, the term *data dimension* denotes a data dimension of a large array, and the term *loop level* denotes a loop that iterates a data dimension of a large array.

For each loop level starting from the outermost (i.e., level 1), *MultiLevelFusion* determines loop fusion at a given level in three steps. The first step tries loop fusion for each data dimension and picks the data dimension that would have the smallest number of fused loops. The second step applies loop fusion for the chosen data dimension. Note that loops of the current level that traverse other data dimensions are also fused if they access the same data dimension. The third step recursively applies *MultiLevelFusion* at a higher level for each fused loop generated at the current level. Since loops can be fused on a data dimension other than the chosen dimension, the dimension s in this step is not always the dimension s' found in the second step of the algorithm.

Since at all data dimensions are examined at most once at each loop level, the cost of *MultiLevelFusion* is $O(D^2 * M)$, where D is the number of data dimensions and M is the cost of *SingleLevelFusion* given in Figure 2.7.

After loop levels and array dimensions are ordered, one issue that still remains is to choose between the choice of fusing two loops and that of embedding one loop into another. Loop embedding is the equivalent of statement embedding in a multi-dimensional program. The resolution is as follows. Given two loops, if they iterate the same data dimension, loop fusion is applied. If, however, the data dimensions iterated by one loop is a subset of data dimensions iterated by another, the former loop is embedded into the latter. In all other cases, two loops are considered as not sharing data, and neither loop fusion nor statement embedding is attempted.

2.4 Optimal Loop Fusion

The maximal fusion presented in the previous section is not optimal because it does not minimize the reuse distance within a fused loop and the data sharing among fused loops. This section formulates these two problems, examines their complexity, and discusses special cases that are polynomial-time solvable.

2.4.1 Loop Fusion for Minimal Reuse Distance

The alignment of loops during fusion determines the reuse distance in the fused loop. The problem for finding the minimal reuse distance can be formulated as a scheduling problem, defined as follows.

> **Problem 2.1 Scheduling for Minimal Live Range (MLR)** is a triple of $(D = (V, E), A)$, where D is a directed acyclic graph (dag) with a vertex set V and edge set E, and A is a set of variables. Each node is in fact an operation which accesses a subset of A. The task is to schedule all operations at some time slot. The correctness of the schedule is specified by the directed edges of the dag. For each edge, the sink node cannot be scheduled until w time slots after the execution of the source node. The quality of the schedule is measured by the live range of variables. Given a schedule, the live range of a variable is the time difference between its first and last use in the schedule.

```
MultiLevelFusion(S: set of data dimensions,
                 L: current loop level)

  /* Step 1. find the best data dimension for loop level L */
  for each data dimension s
    LoopInterchange(s, L)
    apply SingleLevelFusion and count the number of fused loops
  end for

  chose the data dimension s' that has the fewest fused loops

  /* Step 2. fuse loops for level L */
  LoopInterchange(s', L)
  apply loop fusion at level L by invoking SingleLevelFusion

  /* Step 3. for each loop of level L, continue fusion at level L+1 */
  for each loop nest
    recursively apply MultiLevelFusion(S - {s}), where s is the
        data dimension iterated by the loop of level L
  end for
end MultiLevelFusion



Subroutine LoopInterchange(s: data dimension, L: loop level)
    for each loop nest
      if (loop level t (>=L) iterates data dimension s)
         apply loop interchange to make level t into level L if possible
      end if
    end for
end LoopInterchange
```

Figure 2.8 Algorithm for multi-level fusion

> Given a machine with an unlimited number of processors, the problem of MLR is to find a schedule such that
>
> - (Correctness) for each edge of E, the sink node is scheduled at least w time slots after the source node, where w is the weight of the edge, and
> - (Optimality) the sum of the live range of all variables is minimal.

The problem of scheduling for minimal live range differs from traditional problems of task scheduling because the latter group uses a machine of a fixed number of processors. The classical problem, Precedence Constrained Scheduling (PCS), asks whether a dag of tasks can be scheduled in three machine time slots. PCS is NP-complete because it can be reduced from another NP-complete problem of finding a size-k clique in a graph. Because the reduction relies on the fact that the machine resources are limited, the same proof cannot be applied to MLR.

A possible formulation of MLR is as a graph-partitioning problem where operations scheduled at the same time slot are grouped in the same partition of a graph. The graph-based formulation reflects unlimited resources because each partition can contain arbitrarily many operations. A related partitioning problem is k-way cut, defined as follows.

> **Problem 2.2** Given a graph $G = (V, E)$ where each edge has a unit weight but each node can connect any number of edges. Also designate k nodes in V as terminals. The **k-way cut problem** is to find a set of edges of minimal total weight such that removing these edges renders all k terminals disconnected from each other.

The k-way cut problem is NP-hard, as proved by Dahlhaus et al.[DJP$^+$92]. They showed that k-way cut is NP-hard even when k is equal to 3. Their proof used a reduction from the problem of MAX-cut, which finds a maximal number of edges separating two nodes in a graph.

MLR can be reduced from a problem similar to k-way cut. In particular, given a k-way cut problem, one possible way to convert it into a MLR problem is as follows. First, we create a list of k unit-time, sequential operations. Then for each node in the problem of k-way cut, we create an independent operation that can be executed with any of the sequential operations. For example, node u and v in k-way cut become operation u' and v' in MLR. Finally for each edge in k-way cut, we add a new variable that is accessed by the corresponding operations. For example, if a edge connects u and v, we add a variable t that is accessed by u' and v'. After conversion, every possible schedule corresponds to a k-partitioning and vice versa. Note that this reduction process builds a scheduling problem with an unlimited number of data variables.

The data-sharing relationship in MLR can be modeled by the cross-partition weight in k-way cut. One complication, however, is that the data sharing between operations of far apart time slots contributes a longer live range than does the data sharing between operations of close by time slots. For example, the live range between a variable accessed in the first and third time slots is twice the length of the live range of a variable shared between the first and second time slot. Therefore, the goal of minimization for MLR slightly differs from the k-way cut. For the reduction to be correct, MLR should be reduced from the following modified problem of k-way cut.

> **Problem 2.3 Weigted k-way cut** is a graph $G = (V, E)$. V includes a set of k terminals, $v_1, \ldots, v_k$. Each edge is of unit weight, and each node connects to an arbitrary number of edges. Let p be a partitioning of graph nodes so that v_i (i=1,...,k) are in different partitions. Let $n_{i,j}$ be the number of edges between the nodes of the partition containing v_i and those of the partition containing v_j $(i, j = 1, \ldots, k)$. The problem is to find the k-way partitioning so that the function $\Sigma_{j>i \ and \ i,j=1,\ldots,k}(n_{i,j} * (j - i))$ is minimized.

The modified problem is weighted because the objective function has been changed from minimizing $\Sigma n_{i,j}$ to minimizing $\Sigma(n_{i,j} * (j - i))$. The weighted k-way cut problem is not known to be NP-hard. However, considering the complexity of unweighted k-way cut, the weighted version is not likely to be polynomial-time solvable. Section 2.4.3 will revisit this problem and explore a better formulation of loop fusion.

2.4.2 Loop Fusion for Minimal Data Sharing

Maximal loop fusion is not optimal because it does not minimize the amount of data reloading among fused loops. Therefore, it does not minimize the amount of total memory transfer for the whole program. This section first formulates the problem of loop fusion for minimal memory transfer, then gives a polynomial solution to a restricted form of this problem, and finally proves that the complexity of the unrestricted form is NP-complete. In the process, it also points out the inadequacy of the popular fusion model given by Gao et al.[GOST92] and by Kennedy and McKinley[KM93].

Formulation

Given a sequence of loops accessing a set of data arrays, we can model both the computation and the data in a *fusion graph*. A fusion graph consists of nodes— each loop is a node— and two types of edges—directed edges for modeling data dependences and undirected edges for fusion-preventing constraints. Although this definition of a fusion graph looks similar to that of previous work, the objective of fusion is radically different as stated below.

> **Problem 2.4 Bandwidth-minimal fusion problem**: Given a fusion graph, how can we divide the nodes into a sequence of partitions such that
>
> - (Correctness) each node appears in one and only one partition; the nodes in each partition have no fusion preventing constraint among them; and dependence edges flow only from an earlier partition to a later partition in the sequence,
> - (Optimality) the sum of the number of distinct arrays in all partitions is minimal.

The correctness constraint ensures that loop fusion obeys data dependences and fusion-preventing constraints. Assuming arrays are large enough to prohibit cache reuse among disjoint loops, the second requirement ensures optimality because for each loop, the number of distinct arrays is the number of arrays the loop reads from memory during execution. Therefore, the minimal number of arrays in all partitions means the minimal memory transfer and minimal bandwidth consumption for the whole program.

For example, Figure 2.9 shows the fusion graph of six loops. Assuming that loop 5 and loop 6 cannot be fused, but either of them can be freely fused with any other four loops. Loop 6 depends on loop 5. Without fusion, the total number of arrays in the six loops accessed is 20. The optimal fusion leaves loop 5 alone and fuses all other loops. The number of distinct arrays is 1 in the first partition and 6 in the second, thus the total memory transfer is reduced from 20 arrays to 7.

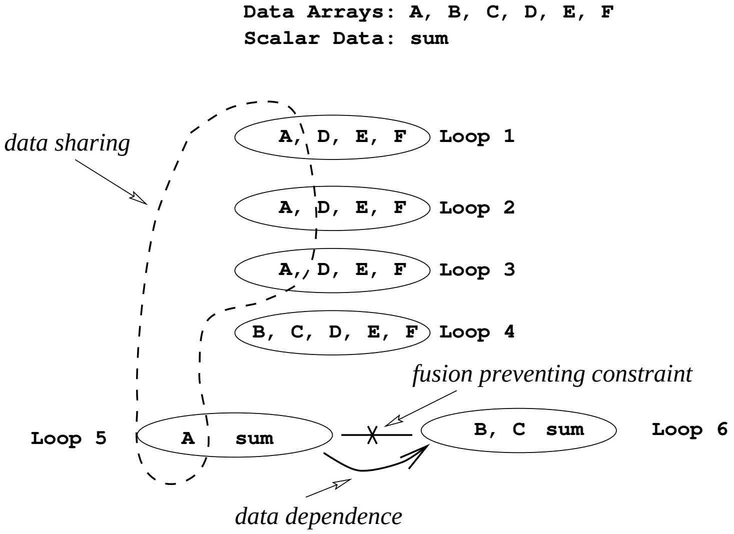


Figure 2.9 Example of bandwidth-minimal loop fusion

The optimality of bandwidth-minimal fusion is different from previous work on loop fusion. Both Gao et al.[GOST92] and Kennedy and McKinley[KM93] constructed a fusion graph in a similar way but modeled data reuse as weighted edges between graph nodes. For example, the edge weight between loop 1 and 2 would be 4 because they share four arrays. Their goal is to partition the nodes so that the total weight of cross-partition edges is minimal.

The sum of edge weights does not correctly model the aggregation of data reuse. For example, in Figure 2.9, loop 1 to 3 each has a single-weight edge to loop 5. But the aggregated reuse between the first three loops and loop 5 should not be 3; on the contrary, the amount of data sharing is 1 because they share access to only one array, A.

To show that weighted-edge formulation is not optimal, it is suffice to give a counter example, which is the real purpose of Figure 2.9. The optimal weighted-edge fusion is to fuse the first five loops and leave loop 6 alone. The total weight of cross-partition edges is 2, which lies between loop 4 and 6. However, this fusion has to load 8 arrays (6 in the first partition and 2 in the second), while the previous bandwidth-minimal fusion needs only 7. Reversely, the total inter-partition edge weight of the bandwidth-minimal fusion is 3, clearly not optimal based on the weighted-edge formulation. Therefore, the weighted-edge formulation does not minimize overall memory transfer.

To understand the effect of data sharing and the complexity of bandwidth-minimal fusion, the remaining part of this section studies a model based on a different type of graphs, hyper-graphs.

Solution Based On Hyper-graphs

The traditional definition of an edge is inadequate for modeling data use because the same data can be shared by more than two loops. Instead, we should use *hyper-edges* because a hyper-edge can connect any number of nodes in a graph. A graph with hyper-edges is called a *hyper-graph*. The optimality requirement of loop fusion can now be restated as follows.

> **Problem 2.5 Bandwidth-minimal fusion problem (II)**: Given a fusion graph as constructed by Problem 2.4, add a hyper-edge for each array in the program, which connects all loops that access the array. How can we divide all nodes into a sequence of partitions such that
>
> - (Correctness) criteria are the same as Problem 2.4, but
> - (Optimality) for each hyper-edge, let the *length* be the number of partitions the edge connects to after partitioning, then the goal is to minimize the total length of all hyper-edges.

The next part first solves the problem of optimal two-partitioning on hyper-graphs and then proves the NP-completeness of multi-partitioning.

Two-partitioning is a special class of the fusion problem where the fusion graph has only one fusion-preventing edge and no data dependence edge among non-terminal nodes. The result of fusion will produce two partitions where any non-terminal node can appear in any partition. The example in Figure 2.9 is a two-partitioning problem.

Two-partitioning can be solved as a connectivity problem between two nodes. Two nodes are connected if there is a path between them. A *path* between two nodes is a sequence of hyper-edges where the first edge connects one node, the last edge connects the other node, and consecutive ones connect intersecting groups of nodes.

Given a hyper-graph with two end nodes, a *cut* is a set of hyper-edges such that taking out these edges would disconnect the end nodes. In a two-partitioning problem, any cut is a legal partitioning. The size of the cut determines the total amount of data loading, which is the total amount of data plus the size of the cut (which is the total amount of data reloading). Therefore, to obtain the optimal fusion is to find a minimal cut.

The algorithm given is Figure 2.10 finds a minimal cut for a hyper-graph. At the first step, the algorithm transforms the hyper-graph into a normal graph by converting each hyper-edge into a node, and connecting two nodes in the new graph when the respective hyper-edges overlap. The conversion also constructs two new end nodes for the transformed graph. The problem now becomes one of finding minimal vertex cut on a normal graph. The second step applies standard algorithm for minimal vertex cut, which converts the graph into a directed graph, splits each node into two and connects them with a directed edge, and finally finds the edge cut set by the standard Ford-Fulkerson method. The last step transforms the vertex-cut to the hyper-edge cut in the fusion graph and constructs the two partitions.

Although algorithm in Figure 2.10 can find minimal cut for hyper-edges with non-negative weights, we are only concerned with fusion graphs where edges have unit-weight. In this case, the first step of the minimal-cut algorithm in Figure 2.10 takes $O(E + V)$; the second step takes $O(V'(E' + V'))$ if breadth-first search is used to find augmenting paths; finally, the last step takes $O(E + V)$. Since $V' = E$ in the second step, the overall cost is $O(E(E' + E) + V)$, where E is the number of arrays, V is the number of loops and E' is the number of the pair of arrays that are accessed by the same loop. In the worst case, $E' = E^2$, and the algorithm takes $O(E^3 + V)$. What is surprising is that although the time is cubic to the number of arrays, it is linear to the number of loops in a program.

By far the solution method has assumed the absence of dependence edges. The dependence relation can be enforced by adding hyper-edges to the fusion graph. Given a fusion graph with N edges and two end nodes s and t, assume the dependence relations form an acyclic graph. Then if node a depends on b, we can add three sets of N edges connecting s and a, a and b, and b and t. Minimal-cut will still find the minimal cut although each dependence adds a weight of N to the total weight of minimal cut. Any dependence violation would add an extra N to the weight of a cut, which makes it impossible to be minimal. In other words, any minimal cut will not place a before b, and the dependence is observed. However, adding such edges would increase the time complexity because the number of hyper-edges will be in the same order as the number of dependence edges.

Input A hyper-graph $G = (V, E)$.
Two nodes s and $t \in V$.

Output A set of edges C, which is a minimal cut between s and t.
Two partitions V_1 and V_2, where $s \in V_1$, $t \in V_2$, $V_1 = V - V_2$, and a edge e connects V_1 and V_2 iff $e \in C$.

Algorithm

```
/* Initialization */
let C, V1 and V2 be empty sets

/* Step 1: convert G to a normal graph */
construct a normal graph G'=(V',E')
   let array map be the one-to-one map between V' and E
   add a node v to V' for each hyper-edge e in E; let map[v] = e
   add edge (v1, v2) in G' iff map[v1] and map[v2] overlap in G

   /* add in two end nodes */
   add two new nodes s' and t' to V'
   for each node v in V'
      add edge (s', v) if map[v] contains s in G
      add edge (t', v) if map[v] contains t in G

/* Step 2: find the minimal vertex cut in G' between s' and t' */
convert G' into a directed graph
split each node in V' and add in a directed edge in between
use For-Fulkerson method to find the minimal edge cut
convert the minimal edge cut into the vertex cut in G'

/* Step 3: construct the cut set and the partitions in G*/
let C be the node cut set of G' found in the previous step
delete all edges of G corresponding to nodes in C
let V1 be the set of nodes connected to s in G; let V2 be V-V1
return C, V1 and V2
```

Figure 2.10 Minimal-cut algorithm for a hyper-graph

The Complexity of General Loop Fusion

Although the two-partitioning problem can be solved in polynomial time, the multi-partitioning form of bandwidth-minimal fusion is NP-complete.

> **Theorem 2.3** Multi-partitioning of bandwidth-minimal fusion is NP-complete when the number of partitions is greater than two.

Proof The fusion problem is in NP because loops or nodes of a fusion graph can be partitioned in a non-deterministic way, and the legality and optimality can be checked in polynomial time.

The fusion problem is also NP-hard. To prove this, we reduce k-way cut problem, defined in Problem 2.2, to the fusion problem. Given a graph $G = (V, E)$ and k nodes to be designated as terminals, k-way cut is to find a set of edges of minimal total weight such that removing the edges renders all k terminals disconnected from each other. To convert a k-way cut problem to a fusion problem, we construct a hyper-graph $G' = (V', E')$ where $V' = V$. We add in a fusion preventing edge between each pair of terminals, and for each edge in E, we add a new hyper-edge connecting the two end nodes of the edge. It is easy to see that a minimal k-way cut in G is an optimal fusion in G' and vice versa. Since k-way cut is NP-complete, bandwidth-minimal fusion is NP-hard when the number of partitions is greater than two. Therefore, it is NP-complete. □

2.4.3 An Open Question

The previous two sections formulated the problem of optimal loop fusion as a graph-partitioning problem, in particular, as unweighted and weighted k-way cut. However, the formulation is not entirely precise for the following two reasons.

On the one hand, optimal loop fusion is simpler than k-way cut because the fusion graph usually has a limited number of edges. The problem of k-way cut assumes that a node can connect to an unbounded number of edges. The number of edges in a fusion graph corresponds to the number of data structures in the program. Although this number is not bounded, it is usually not proportional to the size of the program. Therefore, the cost of optimal k-way cut may not be very high for real programs where the number of data structures is small.

On the other hand, optimal fusion is more complex than k-way cut because of the dependence relations among program statements. Unlike k-way cut in which every node can be grouped with any terminal, a loop can be fused with another only if no dependence is violated. For a real program, a loop can be fused with a subset of loops, and the members of this subset are determined by how other loops are fused.

Considering these two differences, the question therefore remains open on how should loop fusion be formulated and what the complexity is in terms of both the number of loops and data structures.

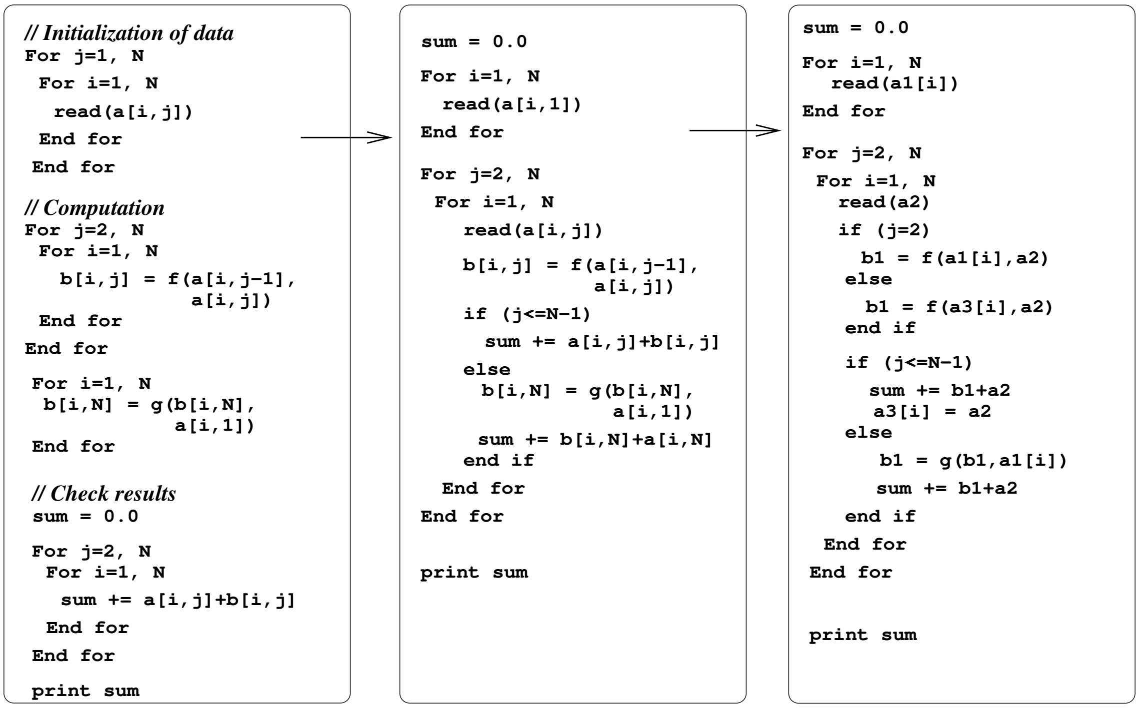
2.5 Advanced Optimizations Enabled by Loop Fusion

Aggressive fusion enables other optimizations. For example, the use of an array can become enclosed within one or a few loops. The localized use allows aggressive storage transformations that are not possible otherwise. This section describes the idea of two such storage optimizations: *storage reduction*, which replaces a large array with a small section or a scalar; and *store elimination*, which avoids writing back new values to an array. Both save a significant more amount of memory bandwidth than loop fusion.

2.5.1 Storage Reduction

After loop fusion, if the live range of an array is shortened to stay within a single loop nest, the array can be replaced by a smaller data section or even a scalar. In particular, two opportunities exist for storage reduction. The first case is where the live range of a data element (all uses of the data element) is short, for example, within one loop iteration. The second case is where the live range spans the whole loop, but only a small section of data elements have such a live range. The first case can be optimized by *array shrinking*, where a small temporary buffer is used to carry live ranges. The second case can be optimized by *array peeling*, where only a reduced section of an array is saved in a dedicated storage. Figure 2.11 illustrates both transformations.

The example program in Figure 2.11(a) uses two large arrays $a[N, N]$ and $b[N, N]$. Loop fusion transforms the program into Figure 2.11(b). Not only does the fused loop contain all accesses to both arrays, the definitions and uses of many array elements are very close in computation. The live range of a b-array element is within one iteration of the inner loop. Therefore, the whole b array can be replaced by a scalar $b1$. The live range of an a-array element is longer, but it is still within every two consecutive j iterations. Therefore, array $a[N, N]$ can be reduced into a smaller buffer $a3[N]$,



```
// Initialization of data
For j=1, N
 For i=1, N
  read(a[i,j])
 End for
End for

// Computation
For j=2, N
 For i=1, N
   b[i,j] = f(a[i,j-1],
              a[i,j])
 End for
End for

For i=1, N
 b[i,N] = g(b[i,N],
            a[i,1])
End for

// Check results
sum = 0.0

For j=2, N
 For i=1, N
   sum += a[i,j]+b[i,j]
 End for
End for

print sum
```

(a) Original program

```
sum = 0.0

For i=1, N
  read(a[i,1])
End for

For j=2, N
 For i=1, N
   read(a[i,j])

   b[i,j] = f(a[i,j-1],
              a[i,j])

   if (j<=N-1)
     sum += a[i,j]+b[i,j]
   else
     b[i,N] = g(b[i,N],
                a[i,1])

     sum += b[i,N]+a[i,N]
   end if
 End for
End for

print sum
```

(b) After loop fusion

```
sum = 0.0

For i=1, N
  read(a1[i])
End for

For j=2, N
 For i=1, N
   read(a2)
   if (j=2)
     b1 = f(a1[i],a2)
   else
     b1 = f(a3[i],a2)
   end if

   if (j<=N-1)
     sum += b1+a2
     a3[i] = a2
   else
     b1 = g(b1,a1[i])
     sum += b1+a2
   end if
 End for
End for

print sum
```

(c) After array shrinking and peeling

Figure 2.11 Array shrinking and peeling

which carries values from one j iteration to the next. A section of $a[N, N]$ array has a live range spanning the whole loop because $a[1 \ldots N, 1]$ is defined at the beginning and used at the end. These elements can be peeled off into a smaller array $a1[N]$ and saved throughout the loop. After array shrinking and peeling, the original two arrays of size N^2 have been replaced by two arrays of size N plus two scalars, achieving a dramatic reduction in storage space.

Storage reduction directly reduces the bandwidth consumption between all levels of memory hierarchy. First, the optimized program occupies a smaller amount of memory, resulting in less memory-CPU transfer. Second, it has a smaller footprint in cache, increasing the chance of cache reuse. When an array can be reduced to a scalar, all its uses can be completed in a register, eliminating cache-register transfers as well.

2.5.2 Store Elimination

While storage reduction optimizes only localized arrays, the second transformation, *store elimination*, improves bandwidth utilization of arrays whose live range spans multiple loop nests. The transformation first locates the loop containing the last segment of the live range and then finishes all uses of the array so that the program no longer needs to write new values back to the array.

```
For i=1, N
 res[i] = res[i]+data[i]
End for

sum = 0.0
For i=1, N
 sum += res[i]
End for

print sum
```

(a) Original program

```
sum = 0.0
For i=1, N
 res[i] = res[i]+data[i]
 sum += res[i]
End for

print sum
```

(b) After loop fusion

```
sum = 0.0

For i=1, N
 sum += res[i]+data[i]
End for

print sum
```

(c) After store elimination

Figure 2.12 Store elimination

The program in Figure 2.12 illustrates this transformation. The first loop in Figure 2.12(a) assigns new values to the *res* array, which is used in the next loop. After the two loops are fused in (b), the writeback of the updated *res* array can be eliminated because all uses of *res* are already completed in the fused loop. The program after store elimination is shown in Figure 2.12(c).

The goal of store elimination differs from all previous cache optimizations because it changes only the behavior of data writebacks and it does not affect the performance of memory reads at all. Store elimination has no benefit if memory latency is the main performance constraint. However, if the bottleneck is memory bandwidth, store elimination becomes extremely useful because reducing memory writebacks is as important as reducing memory reads. The following experiment verifies the benefit of store elimination on two of today's fastest machines: HP/Convex Exemplar and SGI Origin2000 (with R10K processors).

The table in Figure 2.13 lists the reduction in execution time by loop fusion and store elimination. Fusion without store elimination reduces running time by 31% on Origin and 13% on Exemplar; store elimination further reduces execution time by

27% on Origin and 33% on Exemplar. The combined effect is a speedup of almost 2 on both machines, clearly demonstrating the benefit of store elimination.

machines	original	fusion only	store elimination
Origin2000	0.32 sec	0.22 sec	0.16 sec
Exemplar	0.24 sec	0.21 sec	0.14 sec

Figure 2.13 Effect of store elimination

2.6 Summary

The central task of chapter is to minimize the distance of data reuse. It first used the ideal reuse-driven execution to measure the potential of global computation fusion. Then it developed a new fusion algorithm, maximal loop fusion, which fuses all data-sharing program statements whenever possible and achieves bounded reuse distance within a fused loop. The new algorithm employs statement embedding, loop alignment, and iteration reordering to support single-level loop fusion. For programs with multi-dimensional loops and arrays, the new algorithm always minimizes the number of outer loops. Under reasonable assumptions, the time complexity of maximal fusion is $O(V * V' * A * D^2)$, where V is the number of program statements before fusion, V' is the number of fused loops after fusion, A is the number of data arrays, and D is the number of dimensions of the largest array in a program.

Maximal fusion is not optimal because it does not minimize reuse distance within a fused loop and it does not minimize the amount of data sharing among fused loops. The chapter formulated the first problem as weighted k-way cut and the second problem as unweighted k-way cut. It used hyper-graphs to model data sharing and proved that fusion for minimal data sharing is NP-complete.

Loop fusion enables advanced storage optimizations. This chapter described two: storage reduction reduces the size of arrays by array shrinking and array peeling, and store elimination removes memory writebacks by finishing all uses of the data in advance. Store elimination is the first program transformation in the literature that exclusively targets memory bandwidth. These two techniques are not fully developed here and are part of the future work.

One main reason that loop fusion and storage optimizations become profitable on modern machines is that their additional instruction overhead is compensated by fast processors. In general, the dramatically increased computing power has allowed much more aggressive ways of program optimization. However, one question we should not forget to ask is whether we want to optimize programs manually or automatically. Loop fusion is an example of complex program transformation that can and should be automated by a compiler.

In addition to instruction overhead, loop fusion has a side effect on memory hierarchy performance because it may merge too much data access in a fused loop. The next chapter will show how to mitigate this problem by optimizing data layout and exploiting spatial reuse among the global data.

Chapter 3

Global Data Regrouping

3.1 Introduction

Since cache consists of non-unit cache blocks, sufficient use of cache blocks becomes critically important because low cache-block utilization leads directly to both low memory-bandwidth utilization and low cache utilization. For example, for cache blocks of 16 numbers, if only one number is useful in each cache block, 15/16 or 94% of memory bandwidth is wasted, and furthermore, 94% of cache space is occupied by useless data and only 6% of cache is available for data reuse.

A compiler can improve cache-block utilization, or equivalently, cache-block spatial reuse, by packing useful data into cache blocks so that all data elements in a cache block are consumed before it is evicted. Since a program employs many data arrays, the useful data in each cache block may come from two sources: the data within one array, or the data from multiple arrays. Cache-block reuse within a single array is not always possible because not all access to an array can be made contiguous. Common examples are programs with regular, but high dimensional data, and programs with irregular and dynamic data. Furthermore, even in the case of contiguous access within single arrays, cache reuse can still be seriously hindered by excessive cache interference when too many arrays are accessed simultaneously, as for example, after loop fusion. Therefore, a compiler needs to combine useful data from multiple arrays to address the limitations of single-array data reuse.

This chapter presents *inter-array data regrouping*, a global data transformation that first splits and then selectively regroups all data arrays in a program. Figure 3.1 gives an example of this transformation. The left-hand side of the figure shows the example program, which traverses a matrix first by rows and then by columns. One of the loops must access non-contiguous data and cause low cache-block utilization because only one number in each cache block is useful. Inter-array data regrouping combines the two arrays by putting them into a single array that has an extra dimension, as shown in the right-hand side of Figure 3.1. Assuming that the first data dimension is contiguous in memory, the regrouped version guarantees at least two useful numbers in each cache block regardless of the order of traversal.

```
Array a[N,N], b[N,N]

// row-by-row traversal
For j=1, N
  For i=1, N
    F( a[i,j], b[i,j] )
  End for
End for

// column-by-column traversal
For i=1, N
  For j=1, N
    G( a[i,j], b[i,j] )
  End for
End for
```

```
Array c[2,N,N]

// row-by-row traversal
For j=1, N
  For i=1, N
    F( c[1,i,j], c[2,i,j] )
  End for
End for

// column-by-column traversal
For i=1, N
  For j=1, N
    G( c[1,i,j], c[2,i,j] )
  End for
End for
```

Figure 3.1 Example of inter-array data regrouping

In addition to improving cache spatial reuse, data regrouping also reduces the page-table (TLB) working set of a program because it merges multiple arrays into a single one. On modern machines, the cost of TLB overflow is very harmful to performance because CPU cannot continue program execution during a TLB miss.

Inter-array data regrouping can also improve communication performance of shared-memory parallel machines. On these machines, cache blocks are the basis of data consistency and consequently the unit of communication among parallel processors. Good cache-block utilization enabled by inter-array data regrouping can amortize the latency of communication and fully utilize communication bandwidth. However, the use of data regrouping on parallel machines is outside the scope of this dissertation.

The rest of the chapter formulates the problem of inter-array data regrouping, presents its solution and discusses its extensions.

3.2 Program Analysis

Given a program, a compiler identifies in two steps all opportunities of inter-array data regrouping. The first step partitions the program into a sequence of computation phases. A computation phase is defined as a segment of the program that accesses data larger than cache. A compiler can estimate the amount of data access in loop structures. The related compiler analysis techniques will be discussed in Chapter 5, which designs a data analysis tool that estimates the total amount of memory transfer. In places of insufficient information, a compiler can assume that the unknown loop counts are large and unknown data references iterate the whole array. The purpose of these conservative assumptions is to guarantee profitability. The correctness is not affected regardless of compiler assumptions.

The second step of the analysis identifies the sets of compatible arrays. Two arrays are compatible if their sizes differ by at most a constant, and if they are always accessed in the same order in each computation phase. For example, the size of array $A(3, N)$ is compatible with $B(N)$ and with $B(N-3)$ but not with $C(N/2)$ or $D(N, N)$. The access order from $A(1)$ to $A(N)$ is compatible with $B(1)$ to $B(N)$ but not with the order from $C(N)$ to $C(1)$ or from $D(1)$ to $D(N/2)$. The second criterion does allow compatible arrays to be accessed differently in different computation phases, as long as they have the same traversal order in the same phase[5].

The second step requires identifying the data access order within each array. Regular programs can be analyzed with various forms of array section analysis. For irregular or dynamic programs, a compiler can use the data-indirection analysis described in Section 4.3.2 of Chapter 4.

The other important task of the second step is the separation of arrays into the smallest possible units, which is done by splitting constant-size data dimensions into multiple arrays. For example, $A(2, N)$ is converted into $A1(N)$ and $A2(N)$.

After the partitioning of computation phases and compatible arrays, the task of data regrouping becomes clear. First, data regrouping transforms each set of compatible arrays separately because grouping incompatible arrays is either impossible or too costly. Second, a program is now modeled as a sequence of computation phases, each of which accesses a subset of compatible arrays. The goal of data regrouping is to divide the set of compatible arrays into a set of new arrays such that the overall cache-block reuse is maximized in all computation phases.

---

[5] In general, the traversal orders of two arrays need not to be the same as long as they maintain a consistent relationship. For example, array A and B have consistent traversal order if whenever $A[i]$ is accessed, $B[f(i)]$ is accessed, where $f(x)$ is a one-to-one function.

3.3 Regrouping Algorithm

3.3.1 One-Level Regrouping

This section illustrates the problem and the solution of data regrouping through an example—the application *Magi* from DOD, which simulates the shock and material response of particles in a three-dimensional space (based on smoothed particle hydrodynamics method). The table in Figure 3.2 lists the six major computation phases of the program as well as the attributes of particles used in each phase. Since the program stores an attribute of all particles in a separate array, different attributes do not share the same cache block. Therefore, if a computation phase uses k attributes, it needs to load in k cache blocks when it accesses a particle.

Computation phases		Attributes accessed
1	*constructing interaction list*	`position`
2	*smoothing attributes*	`position, speed, heat, derivate, viscosity`
3	*hydrodynamic interactions 1*	`density, momentum`
4	*hydrodynamic interactions 2*	`momentum, volume, energy, cumulative totals`
5	*stress interaction 1*	`volume, energy, strength, cumulative totals`
6	*stress interaction 2*	`density, strength`

Figure 3.2 Computation phases of a hydrodynamics simulation program

Combining multiple arrays can reduce the number of cache blocks accessed and consequently improve cache-block reuse. For example, we can group *position* and *velocity* into a new array such that the ith element of the new array contains the position and velocity of the ith particle. After array grouping, each particle reference of the second phase accesses one fewer cache blocks since *position* and *velocity* are now loaded by a single cache block. In fact, we can regroup all five arrays used in the second phase and consequently merge all attributes into a single cache block (assuming a cache block holds five attributes).

However, excessive grouping in one phase may hurt cache-block reuse in other phases. For example, grouping *position* with *speed* wastes a half of each cache block in the first phase because the *speed* attribute is never referenced in that phase.

The example program shows two requirements for data regrouping. The first is to fuse as many arrays as possible in order to minimize the number of loaded cache blocks, but at the same time, the other requirement is not to introduce any useless data through regrouping. In fact, the second requirement mandates that two arrays should not be grouped unless they are always accessed together. Therefore, the goal of data regrouping is to partition data arrays such that (1) two arrays are in the same partition only if they are always accessed together, and (2) the size of each partition is the largest possible. The first property ensures no waste of cache, and the second property guarantees the maximal cache-block reuse.

Although condition (1) might seem a bit restrictive in practice, many applications use multiple fields of a data structure array together. The algorithm will split each field as a separate array. In addition, aggressive loop fusion often gathers data access of a large number of arrays in a fused loop. Therefore, it should be quite common for two or more arrays to always be accessed together. Later, Section 3.4 discusses methods for relaxing condition (1) at the cost of making the analysis more complex.

The problem of optimal regrouping is equivalent to a set-partitioning problem. A program can be modeled as a set and a sequence of subsets where the set represents all arrays and each subset models the data access of a computation phase in the program.

Given a set and a sequence of subsets, we say two elements are *buddies* if for any subset containing one element, it must contain the other one. The *buddy* relation is reflexive, symmetric, and transitive; therefore it is a partition. A buddy partitioning satisfies the two requirements of data regrouping because (1) all elements in each partition are buddies, and (2) all buddies belong to the same partition. Thus the data-regrouping problem is the same as finding a partitioning of buddies. For example in Figure 3.2, array *volume* and *energy* are buddies because they are always accessed together.

The buddy partitioning can be solved with efficient algorithms. For example, the following partitioning method uses set memberships for each array, that is, a bit vector whose entry i is 1 if the array is accessed by the ith phase. The method uses a radix sort to find arrays with the same set memberships, i.e. arrays that are always accessed together. Assuming a total of N arrays and S computation phases, the time complexity of the method is $O(N * S)$. If a bit-vector is used for S in the actual implementation, the algorithm runs in $O(N)$ vector steps. In this sense, the cost of regrouping is linear to the number of arrays.

3.3.2 Optimality

Qualitatively, the algorithm groups two arrays when and only when it is always profitable to do so. To prove, consider on the one hand, data regrouping never includes any useless data into cache, so it is applied only when profitable; on the other hand, whenever two arrays can be merged without introducing useless data, they are regrouped by the algorithm. Therefore, *data regrouping exploits inter-array spatial reuse when and only when it is always profitable.*

Under reasonable assumptions, the optimality can also be defined quantitatively in terms of the amount of memory access and the size of TLB working set. The key link between an array layout and the overall data access is the concept called *iteration footprint*, which is the number of distinct arrays accessed by one iteration of a computation phase. Assuming an array element is smaller than a cache block but an array is larger than a virtual memory page, then the iteration footprint is equal to the number of cache blocks and the number of pages accessed by one iteration. The following lemma shows that data regrouping minimizes the iteration footprint.

> **Lemma 3.1** Under the restriction of no useless data in cache blocks, data regrouping minimizes the iteration footprint of each computation phase.

Proof After buddy partitioning, two arrays are regrouped when and only when they are always accessed together. In other words, two arrays are combined when and only when doing so does not introduce any useless data. Therefore, for any computation phase after regrouping, no further array grouping is possible without introducing useless data. Thus, the iteration footprint is minimal after data regrouping. □

The size of a footprint directly affects cache performance because the more arrays are accessed, the more active cache blocks are needed in cache, and therefore, the more chances of premature eviction of useful data caused by either limited cache capacity or associativity. For convenience, we refer to both cache capacity misses and cache interference misses collectively as *cache overhead misses*. It is reasonable to assume that the number of cache overhead misses is a non-decreasing function on the number of active arrays. Intuitively, a smaller footprint should never cause more overhead misses because a reduced number of active cache blocks can always be arranged so that their conflicts with cache capacity and with each other do not increase. With this assumption, the following theorem proves that a minimal footprint leads to minimal cache overhead.

> **Theorem 3.1** Given a program of n computation phases, where the total number of cache overhead misses is a non-decreasing function on the size of its iteration footprint k, then data regrouping minimizes the total number of overhead misses in the whole program.

Proof Assuming the number of overhead misses in the n computation phases is $f_1(k_1), f_2(k_2), \ldots, f_n(k_n)$, then the total amount of memory re-transfer is proportional to $f_1(k_1) + f_2(k_2) + \ldots + f_n(k_n)$. According to the previous lemma, $k_1, k_2, \ldots, k_n$ are the smallest possible after regrouping. Since all functions are non-decreasing, the sum of overhead misses is therefore minimal after data regrouping. $\square$

The assumption made by the theorem covers a broad range of data access patterns in real programs, including two extreme cases. The first is the worst extreme, where no cache reuse happens, for example, in random data access. The total number of cache misses is linear to the size of the iteration footprint since each data access causes a cache miss. The other extreme is perfect cache reuse where no cache overhead miss occurs, for example, in contiguous data access. The total number of repeated memory transfer is zero. In both cases, the number of cache overhead misses is a non-decreasing function on the size of the iteration footprint. Therefore, data regrouping is optimal in both cases according to the theorem just proved.

In a similar way, data regrouping minimizes the overall TLB working set of a program. Assuming arrays do not share the same memory page, the size of the iteration footprint, i.e. the number of distinct arrays accessed by a computation phase, is in fact the size of its TLB working set. Since the size of TLB working set is a non-decreasing function over the iteration footprint, the same proof can show that data regrouping minimizes the overall TLB working set of the whole program.

A less obvious benefit of data regrouping is the elimination of useless data by grouping only those parts that are used by a computation phase of a program. The elimination of useless data by array regrouping is extremely important for applications written in languages with data abstraction features, as in, for example, C, C++, Java and Fortran 90. In these programs, a data object contains many attributes, but only a fraction of them is used in a given computation phase. Data regrouping will break each attribute into a separate location and group only those that are used in a way that is compile-time optimal for the whole program.

In summary, the regrouping algorithm is optimal because it minimizes all iteration footprints of a program. With the assumption that cache overhead is a non-decreasing function over the size of iteration footprints, data regrouping achieves maximal cache reuse and minimal TLB working set.

3.3.3 Multi-level Regrouping

The previous sections have been aimed at improving cache-block reuse and therefore did not group data at granularity larger than an array element. This section overcomes this limitation by grouping arrays at higher levels. The extension is beneficial because optimizing the layout of array segments reduces cache interference and the page-table working set.

The example program in Figure 3.3 illustrates multi-level data regrouping. Array A and B are grouped at the element level to improve spatial reuse in cache blocks. In addition, the columns of all three arrays are grouped so that each outer-loop iteration accesses a contiguous segment of memory. Consider, for example, the data access of the first iteration of the outer loop. The first inner loop iterates through the first column $D[1-2, 1 \ldots N, 1, 1]$. Then the second inner loop traverses through the second column $D[1 \ldots N, 2, 1]$. Therefore, each outer-loop iteration accesses a contiguous section of memory. Indeed, multi-level regrouping achieves contiguous data access even for non-perfectly nested loops.

It should be noted that popular programming languages such as Fortran do not allow arrays of non-uniform dimensions like those of array D. However, this is not a problem when regrouping is applied by a back-end compiler. In addition as revealed later in the evaluation chapter, source-level regrouping may negatively affect register allocation of the back-end compiler. However, data regrouping does not change the relationship of temporal reuse of any variable. The problem must be the confusion caused solely by source-level changes. Therefore, the problem should be easily solved if data regrouping is applied by the back-end compiler itself.

The algorithm for multi-level regrouping is shown in Figure 3.4. The first step of *MultiLevelRegrouping* collects simultaneous data access at all array dimensions. Two criteria are used to find the set of arrays accessed at a given dimension. The first is necessary for the algorithm to be correct. The second criterion does not affect correctness, but it make sure that the algorithm considers only those memory references that access the whole array. The sets of arrays found for each data dimension correspond

```
for i
  for j
    g( A[j,i], B[j,i] )
  end for
  for j
    t( C[j,i] )
  end for
end for
```

```
A[j,i] -> D[1,j,1,i]
B[j,i] -> D[2,j,1,i]
C[j,i] ->   D[j,2,i]
```

Figure 3.3 Example of multi-level data regrouping

to the computation phases for that dimension. The second step of the algorithm then applies one-level regrouping for each data dimension. Subroutine *OneLevelRegrouping* uses the partitioning algorithm discussed in Section 3.3.1 to regroup arrays at a single data dimension. The correctness of multi-level regrouping is proved in the following theorem. The purpose of the proof is to show that the grouping decision at a lower level (e.g., grouping array a with b) does not contradicts the decision at a higher level (e.g., separating a and b).

> **Theorem 3.2** If the algorithm in Figure 3.3 merges two arrays at data dimension d, the algorithm must also group these two arrays at all dimensions higher than d.

Proof It suffices to prove that if the algorithm groups two arrays at d, the two arrays are always accessed together at dimensions higher than d. Suppose a loop l exists where the two arrays are not accessed together at an outer dimension. Among all references (of these two arrays) that are considered for dimension d, some of them must be enclosed by loop l because l iterates through a dimension higher than d. Since the two arrays are not always accessed together under loop l, the first step of the algorithm must find a set for dimension d that contains only one of the two arrays. Therefore, the two arrays will not be grouped at dimension d by the algorithm. Contradiction. □

3.4 Extensions

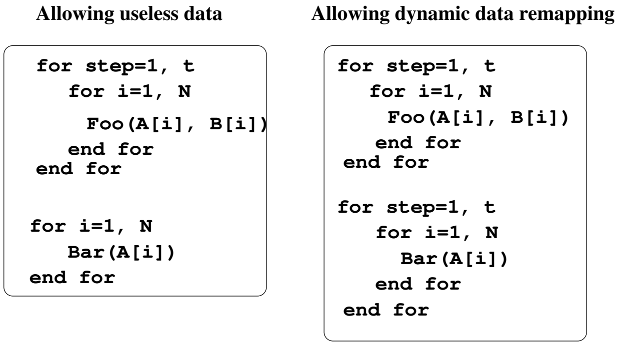
The previous section makes two restrictions in determining data regrouping. The first is disallowing any useless data, and the second is assuming a static data layout without dynamic data re-mapping. This section relaxes these two restrictions and gives

```
Assumptions
   arrays are stored in column-major order
   only memory references to compatible arrays are considered

MutiLevelRegrouping
   /* Step 1. find the subsets of arrays accessed in all loop levels */
   for each loop i
       examine all array references inside this loop, and
       for each data dimension d, find the set s of arrays such that
            (1) each array of s is accessed at all dimensions higher than or
                equal to d by loop i and its outer loops, and
            (2) each array of s is accessed at all other dimensions by the
                inner loops of i
   end for

   /* Step 2. partition arrays for each data dimension */
   for each data dimension d
       let S be the collection of sets found in Step 2 for dimension d
       let A be the set of all arrays
       OneLevelRegrouping(A, S, d)
   end for
end MultiLevelRegrouping


OneLevelRegrouping(A: all arrays, S: subsets of A, d: current dimension)
   let N be the size of A

   /* construct a bit vector for each subset */
   for each subset s in S
       construct a bit vector b of length N
       for i from 1 to N
           if (array i is in s) b[i]=1 otherwise b[i]=0
       end for
   end for

   /* partition arrays */
   sort all bit vectors using radix sort
   group arrays that have the same bit vector at dimension d
end OneLevelRegrouping
```

Figure 3.4 Algorithm for multi-level data regrouping modified solutions. In addition, this section expands the scope of data regrouping to minimizing not only memory reads but also memory writebacks.

3.4.1 Allowing Useless Data

Sometimes allowing useless data may lead to better performance. An example is the first program in Figure 3.5. Since the first loops is executed 100 times more often than the second loop, it is very likely that the benefit of grouping A and B in the first loop exceeds the overhead of introducing useless data in the second loop.



Allowing useless data

```
for step=1, t
   for i=1, N
     Foo(A[i], B[i])
   end for
end for

for i=1, N
   Bar(A[i])
end for
```

Allowing dynamic data remapping

```
for step=1, t
   for i=1, N
    Foo(A[i], B[i])
   end for
end for

for step=1, t
   for i=1, N
     Bar(A[i])
   end for
end for
```

Figure 3.5 Examples of extending data regrouping

However, the tradeoff depends on the exact performance gain due to data regrouping and the performance loss due to useless data. Both the benefit and cost are machine dependent. Therefore, not to include useless data is in fact compile-time optimal because otherwise regrouping cannot guarantee profitability. In practice, the implementation of data regrouping considers only frequently executed computation phases. It applies data regrouping only on loops that are inside a time-step loop.

When the exact run-time benefit of regrouping and the overhead of useless data are known, the problem of optimal regrouping can be formulated with a weighted, undirected graph, called a *data-regrouping graph*. Each array is a node in the graph. The weight of each edge is the run-time benefit of regrouping its two end nodes minus the overhead of such grouping. The goal is to pack arrays that are most beneficial into the same cache block. However, the packing problem on a data-regrouping graph is NP-hard because it can be reduced from the G-partitioning problem[KH78].

3.4.2 Allowing Dynamic Data Regrouping

Until now, data regrouping uses a single data layout for the whole program. An alternative strategy is to allow dynamic regrouping of data between computation phases so that the data layout of a particular phase can be optimized without worrying about the side effects in other phases. An example is the program in the right-hand side of Figure 3.5. The best strategy may be to group A and B at the beginning of the program and then separate these two arrays after the first time-step loop.

As in the case of allowing useless data, the profitability of dynamic regrouping depends on the exact benefit of data grouping and the overhead of run-time readjustment, both of which are machine dependent. Therefore, not to use dynamic regrouping is compile-time optimal because it never causes negative performance impact. A possible extension is to apply data regrouping within different time-step loops and insert dynamic data regrouping in between.

When the precise benefit of regrouping and the cost of dynamic re-mapping is known, the problem can be formulated in the same way as the one given by Kremer[Kre95]. In his formulation, a program is separated into computation phases. Each data layout results in a different execution time for each phase plus the cost of changing data layouts among phases. The optimal layout, either dynamic or static, is the one that minimizes the overall execution time. Without modification, Kremer's formulation can model the search space of all static or dynamic data-regrouping schemes. However, as Kremer has proved, finding the optimal layout is NP-hard. Since the search space is generally not large, he successfully used 0-1 integer programming to find the optimal data layout. The same method can be used to find the optimal data regrouping when dynamic regrouping is allowed.

3.4.3 Minimizing Data Writebacks

On machines with insufficient memory bandwidth, data writebacks impede memory read performance because they consume part of the available memory bandwidth. To avoid unnecessary writebacks, read-only data should not be in the same cache block as modified data otherwise read-only data will be unnecessarily written back. Therefore, data regrouping should not combine arrays unless they are all read-only or all modified in any computation phase. This new requirement can be easily enforced as follows. For each computation phase, split the accessed arrays into two disjoint subsets: the first is the set of read-only arrays and the second is the modified arrays.

Treat each subset as a distinctive computation phase and then apply the partitioning. As a result, two arrays are grouped if and only if they are always accessed together, and the type of the access is either both read-only or both modified. With this extension, data regrouping finds the largest subsets of arrays that can be grouped without introducing useless data or redundant writebacks. Note that two arrays are grouped does not mean they must be read-only or modified throughout the whole program. They can be both read-only in some phases and both modified in other phases.

The above extension can be easily included into the multi-level data regrouping algorithm given in Figure 3.4. In its first step, if the data dimension is the innermost dimension, the algorithm will split each set s into two sets, one with all read-only arrays and one with modified arrays. The regroupings at higher dimensions are not affected because they cannot interleave data into the same cache block. All other aspects of the algorithm are also unchanged.

When redundant writebacks are allowed, data regrouping can be more aggressive by first combining data solely based on data access and then separating read-only and modified data within each partition. The separation step is not easy because different computation phases read and write a different set of arrays. The general problem can be modeled with a weighted, undirected graph, in which each array is a node and each edge has a weight labeling the combined effect of both regrouping and redundant writebacks. The goal of regrouping is to pack nodes into cache blocks to maximize the benefit. As in the case of allowing useless data, the packing problem here is also NP-hard because it can be reduced from the G-partitioning problem[KH78].

3.5 Summary

This chapter has developed inter-array data regrouping, a global data transformation that first splits and then regroups all arrays to achieve maximal inter-array spatial reuse. The compiler first divides a program into computation phases and then partitions arrays into compatible sets. Data regrouping is applied within a compatible set. Two arrays are grouped if and only if they are always accessed together. The algorithm for regrouping is efficient; its time complexity is $O(V * A)$, where V is the length of the program and A is the number of data structures in the program. In the case of high dimensional data arrays, data grouping is applied at a hierarchy of levels to maximize the degree of contiguous access at each loop level.

The regrouping method is conservative because it never interleaves useful data with useless data at any moment of computation. Therefore it guarantees profitability. When useless data and dynamic regrouping are prohibited, the regrouping algorithm is also optimal. The relaxation of either constraint makes optimal data layout dependent on the exact run-time effect of a data transformation, which makes the result machine dependent. In contrast, the conservative regrouping achieves the best machine-independent data layout, that is, the compile-time optimal solution. In addition, the chapter proved that relaxing either constraint leads to NP-hard problems.

Data regrouping can be extended to avoid unnecessary memory writebacks by separating read-only access and read-write access into different cache blocks. The extension has been incorporated in the overall algorithm of multi-level, inter-array data regrouping.

Global data regrouping enables users to define data structures in their own style without worrying about their impact on performance. Programmers should not attempt data optimizations because the optimal array layout depends on the computation structure of the program. Manual transformation would have to readjust every time the program changes. With inter-array data regrouping described in this chapter, a compiler can now derive the optimal data layout regardless of the initial choice of users. Indeed, data regrouping is a perfect job for an automatic compiler and a compiler can do it perfectly.

Chapter 4

Run-time Cache-reuse Optimizations

4.1 Introduction

The previous chapters have assumed that both the data structure and its access pattern are fixed and are known to a compiler. A large class of applications, however, employs extensible data structures whose shape and content are constructed and changed at run time. An example is molecular dynamics simulation, which models the movement of particles in some physical domain (e.g. a 3-D space). The distribution of molecules remains unknown until run time, and the distribution itself changes during the computation. Another example is sparse linear algebra, where the non-zero entries in a sparse matrix change dynamically. Because of their non-uniform and unpredictable nature, they are called *irregular and dynamic applications.*

Irregular and dynamic applications pose two new problems for cache optimization. First, since a compiler knows neither the data and its access, optimizations cannot be applied at compile-time. In addition, since the computation evolves during the execution, no fixed program organization is likely to perform well at all times. Therefore, a program may have to be transformed multiple times during the execution.

This chapter studies run-time optimizations for improving cache reuse in irregular and dynamic applications. Specifically, it presents the run-time version of computation fusion and data grouping—*locality grouping* and *data packing.* The first half of the chapter describes and evaluates these two transformations. The second half is devoted to the compiler support for dynamic data packing. It first presents the compiler analysis that automatically detects all opportunities of data packing. Since switching among different data layouts at run time carries a significant overhead, two compiler optimizations are then introduced to eliminate most of this overhead.

4.2 Locality Grouping and Data Packing

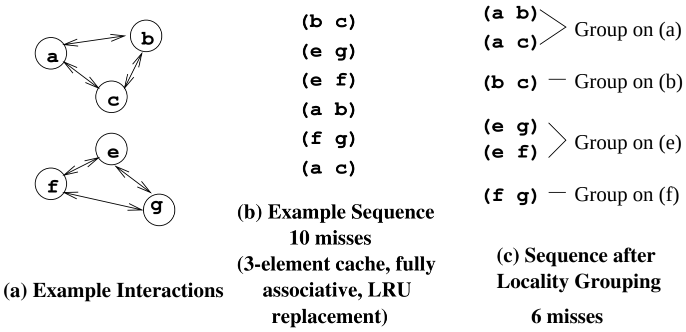
This section describes two run-time transformations: locality grouping, which fuses dynamic computations on the same data item; and dynamic data packing, which then groups data items that are used together. Both transformations are evaluated, individually and combined, through various access sequences on simulated caches.

4.2.1 Locality Grouping

The effectiveness of cache is predicated on the existence of locality and good computation structure exploiting that locality. In a dynamic application such as molecular dynamics simulation, the locality comes directly from its physical model in which a particle interacts only with its neighbors. A set of neighboring particles forms a locality group in which most interactions occur within the group. In most programs, however, locality groups are not well separated. Although schemes such as domain partitioning and space-fitting curve ordering exist for explicitly extracting locality, they are time-consuming and may therefore not be cost-effective in improving cache performance of a sequential execution. Another limitation of previous work is that it relies on user knowledge and manual program transformation. To pursue a faster algorithm and a general program transformation model, this section presents the most efficient, yet also most general reordering scheme, locality grouping.

Given a sequence of independent computations, locality grouping clusters those sharing access to the same data. Figure 4.1(a) shows an example input to a N-body simulation program. Graph (a) draws three example objects and their interactions and Graph (b) is the example sequence of all interactions, which are independent computations. Assuming a cache of 3 objects, the example sequence incurs 10 misses. Locality grouping reorders the sequence so that all computations on the same object are clustered. The new sequence starts with all interactions on object a, then b, until the last object g. The locality-grouped access sequence incurs only 6 misses.

Locality grouping incurs minimal run-time overhead. It consists of a two-pass radix sort: the first pass collects a histogram and the second pass produces the locality-grouped sequence. Locality grouping is widely applicable and can optimize any set of independent computations. A compiler can automate locality grouping by identifying parallel computations and inserting a call to a run-time library. The legality and profitability of locality grouping can be determined either by compiler



(a) Example Interactions

(b) Example Sequence 10 misses (3-element cache, fully associative, LRU replacement)

(c) Sequence after Locality Grouping 6 misses

Figure 4.1 Example of locality grouping

analysis or user directives. One example use of user directive is presented in detail in the second half of this chapter.

The remainder of the section evaluates locality grouping on a data set from *mesh*, a structural simulation. The data set is a list of edges of a mesh structure of some physical object such as an airplane. Each edge connects two nodes of the mesh. This specific data set, provided by the Chaos group at University of Maryland, has 10K nodes and 60K edges. The experiment simulates only the data accesses on a fully associative cache in order to isolate the inherent cache reuse behavior from other factors. In fact, the simulation is very similar to the one used in Section 2.2, which measures the reuse distance of repeated data access. The cache misses are the reuses whose reuse distance is greater than or equal to cache size. The specific cache sizes measured are 2K and 4K objects. The cache uses unit-length cache lines.

Figure 4.2 gives the miss rate of *mesh* with and without locality grouping. Locality grouping eliminates 96.9% of cache misses in the 2K cache and 99.4% in the 4K cache. The miss rates after locality grouping are extremely low, especially in the 4K cache (0.37%). Further decreasing miss rate with more powerful reordering schemes in this case is unlikely to be cost-effective if the overhead of extra execution time does not out-weigh the additional gain.

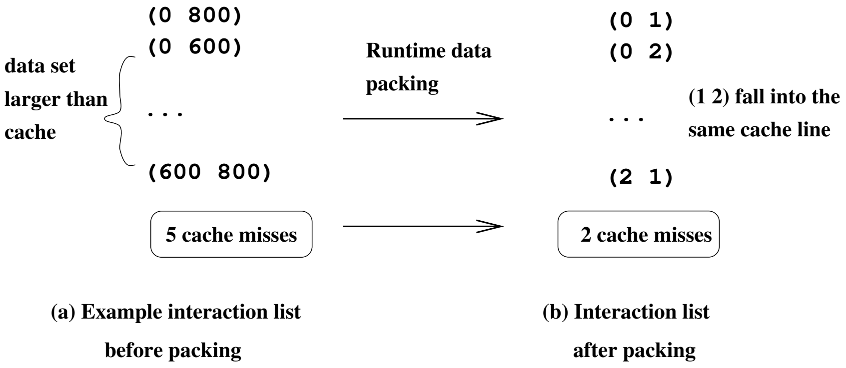
	Original		After locality grouping	
miss rate of *mesh*	2K cache	4K	2K cache	4K
	93.2%	63.5%	2.93%	0.37%

Figure 4.2 Effect of locality grouping

4.2.2 Dynamic Data Packing

Correct data placement is critical to effective use of available memory bandwidth. *Dynamic data packing* is a run-time optimization that groups data accessed at close intervals in the program into the same cache line. For example, if two objects are always accessed consecutively in a computation, placing them adjacent to each other increases bandwidth utilization by increasing the number of bytes on each line that are used before the line is evicted.

Figure 4.3 will be used as an example throughout this section to illustrate the packing algorithms and their effects. Figure 4.3(a) shows an example access sequence. The objects are numbered by their location in memory. In the sequence, the first object interacts with the 600th and 800th object and subsequently the latter two objects interact with each other. Assume that the cache size is limited and the access to the last pair of the 600th and 800th objects cannot reuse the data loaded at the beginning. Since each of these three objects is on different cache lines, the total number of cache misses is 5. A transformed data layout is shown in Figure 4.3(b), where the three objects are relocated at positions 0 to 2. Assuming a cache line can hold three objects, the transformed layout only incurs two cache misses, a significant reduction from the previous figure of 5 misses.



(a) Example interaction list before packing

(b) Interaction list after packing

Figure 4.3 Example of data packing

The rest of this section presents three packing algorithms and a comparison study of their performance on different types of run-time inputs.

Packing Algorithms

The simplest packing strategy is to place data in the order they first appear in the access sequence. I call this strategy *consecutive packing* or *first-touch packing*. The packing algorithm is shown in Figure 4.4. To ensure that each object has one and only one location in the new storage, the algorithm uses a tag for each object to label whether the object has been packed or not.

```
initializing each tag to be false (not packed)
for each object i in the access sequence
    if i has not been packed
       place i in the next available location
       mark its tag to be true (packed)
    end if
end iteration
place the remaining unpacked objects
```

Figure 4.4 Algorithm of consecutive data packing

Consecutive packing carries a minimal time and space overhead because it traverses the access sequence and object array once and only once. For access sequences in which each object is referenced at most once, consecutive packing yields optimal cache line utilization because the objects are visited in stride-one fashion during the computation. Achieving an optimal packing in the presence of repeated accesses, on the other hand, is NP-complete, as this problem can be reduced to the G-partition problem[KH78] following a similar reduction by Thabit[Tha81]. The packing algorithms presented in this section are therefore based on heuristics.

One shortcoming of consecutive packing is that it does not take into account the different reuse patterns of different objects. *Group packing* attempts to overcome this problem by classifying objects according to their reuse pattern and applying consecutive packing within each group. In the example in Figure 4.3(b), the first object is not reused later but the 600th and 800th object are reused after a similar interval. Based on reuse patterns, group packing puts the latter two objects into a new group and packs them separately from the first object. If we assume a cache line of two objects, consecutive packing fails to put the latter two objects into one cache line but group packing succeeds. As a result, consecutive packing yields four misses while group packing incurs only three.

The key challenge for group packing is how to characterize a reuse pattern. The simplest approach is to use the average reappearance distance of each object in the access sequence, which can be efficiently computed in a single pass. More complex characterizations of reuse patterns may be desirable if a user or compiler has additional knowledge on how objects are reused. However, more complex reuse patterns may incur higher computation costs at run time.

The separation of objects based on reuse patterns is not always profitable. It is possible that two objects with the same reuse pattern are so far apart in the access sequence so that they can never be in cache simultaneously. In this case, we do not want to pack them together. To solve this problem, we need to consider the distance between objects in the access sequence as well as their reuse pattern. This consideration motivates the third packing algorithm, *consecutive-group packing*.

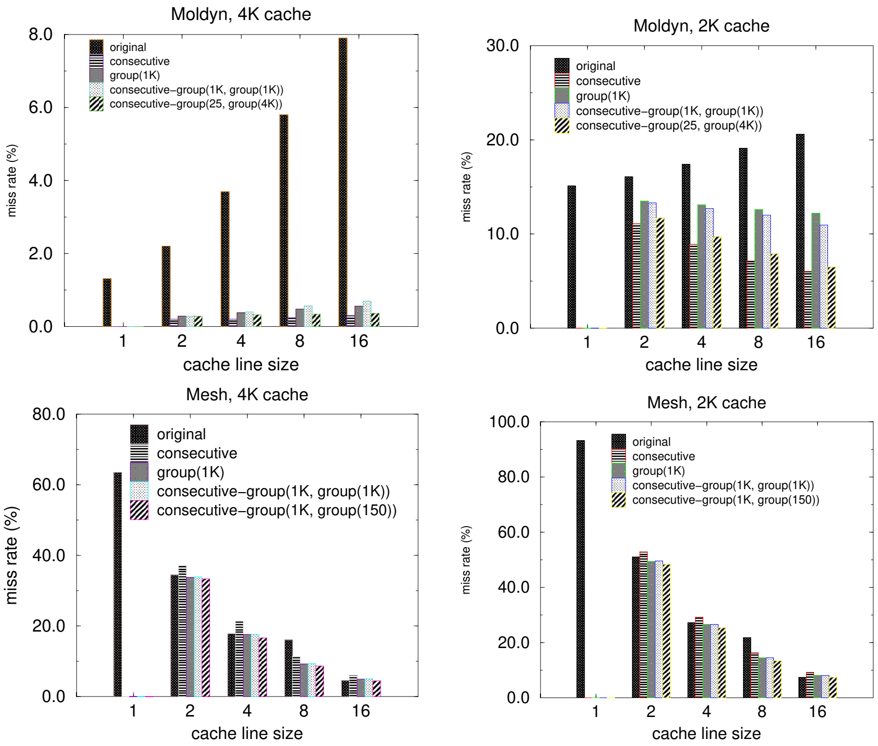
Consecutive-group packing groups objects based on the position of their first appearance. For example, it first groups the objects appeared in the first N positions in the access sequence, then the objects in the next N positions, and so on until the end of the access sequence. The parameter N is the *consecutive range*. Within each range group, objects can then be reorganized with group packing.

The length of the consecutive range determines the balance between exploiting closeness and exploiting reuse patterns. When the consecutive range is 1, data packing is the same as consecutive packing. When the range is the full sequence, the packing is the same as group packing. In this sense, these three packing algorithms are actually one single packing heuristic with different parameters.

Evaluation of Packing Algorithms

All three packing algorithms are evaluated on *mesh* and another input access stream which is extracted from *moldyn*, a molecular dynamics simulation program. The *Moldyn* program initializes approximately 8K molecules with random positions. As before, the experiment simulated only the data access on a fully associative cache.

The group packing classifies objects by their average reappearance distance; it is parameterized by its distance granularity. A granularity of 1000 means that objects whose average reappearance distance fall in each 1000-element range are grouped together. Consecutive-group packing has two parameters: the first is the consecutive range, and the second is the grouping packing algorithm used inside each range.



Moldyn, 4K cache

Moldyn, 2K cache

Mesh, 4K cache

Mesh, 2K cache

Figure 4.5 *Moldyn* and *Mesh*, on 2K and 4K cache

The four graphs in Figure 4.5 show the effect of packing on the *moldyn* and *mesh* data sets. The upper-left graph draws the miss rate on a 4K-sized cache for different cache line sizes from 1 to 16 molecules long. The miss rate of the original data layout, shown by the first bar of each cluster, increases dramatically as cache lines get longer. The cache with 16-molecule cache lines incurs 6 times the number of misses of the unit-line cache. Since the total amount of memory transfer is the number of misses times cache line size, the 16-molecule cache lines result in 96 times the memory transfer volume of the unit cache line case—it is wasting 99% of the available memory bandwidth! Even 2-molecule cache lines waste over 80% of available memory bandwidth. After various packing algorithms are applied, however, the miss rates drop significantly, as indicated in the remaining four bars in each cluster. Consecutive packing reduces the miss rate by factors ranging from 7.33 to over 26. Because of the absence of consistent reuse pattern, group and consecutive-group packing do not perform as well as consecutive packing but nevertheless reduce the miss rate by a similar amount. The upper-right graph shows the effect of packing on a 2K cache, which is very similar to the 4K cache except that the improvement is smaller. Consecutive packing still performs the best and reduces the miss rate by 27% to a factor of 3.2.

The original access sequence of the *mesh* data set has a cyclic reuse pattern and a very high miss rate; see, for example, 64% on the 4K cache, shown in the lower-left graph of Figure 4.5. Interestingly, the cyclic data access pattern scales well on longer cache lines, except at the size of 8. Data packing, however, evenly reduces miss rate on all cache line sizes, including the size of 8. At that size, packing improves from 29% to 46%. On other sizes, consecutive packing and group packing yield slightly higher miss rates than the original data layout. One configuration, consecutive-group(1K,group(150)), is found to be the best of all; it achieves the lowest miss rate in all cases, although it is only marginally better on sizes other than 8. Similar results are seen on a 2K cache, shown by lower-right graph in Figure 4.5. The same version of consecutive-group packing reduces miss rate by 1% to 39%. It should be noted that the result of consecutive-group packing is very close to the ideal case where the miss rate halves when cache line size doubles. As shown in the next section, dynamic packing, when combined with locality grouping, can reduce the miss rate to as low as 0.02%.

4.2.3 Combining Computation and Data Transformation

When locality grouping is combined with data packing on *mesh* (*moldyn* was already in locality-grouped form), the improvement is far greater than when they are individually applied. Figure 4.6 shows miss rates of *mesh* after locality grouping. On a 4K cache, the miss rate on a unit-line cache is reduced from 64% to 0.37% after locality grouping. On longer cache-line sizes, data packing further reduces the miss rate by 15% to a factor of over 6. On the 16-molecule cache line case, the combined effect is a reduction from a miss rate of 4.52% (shown in Figure 4.5) to 0.02%, a factor of 226. On a 2K cache with 16-molecule cache lines, the combined transformations reduce miss rate from 7.48% to 0.25%, a factor of 30. Although not shown in the graph, group and consecutive-group packing do not perform as well as consecutive packing.

In summary, the simulation results show that locality grouping effectively extracts computation locality, and data packing significantly improves data locality. The effect of data packing becomes even more pronounced in caches with longer cache lines. In both programs, simple consecutive packing performs the best after locality grouping,

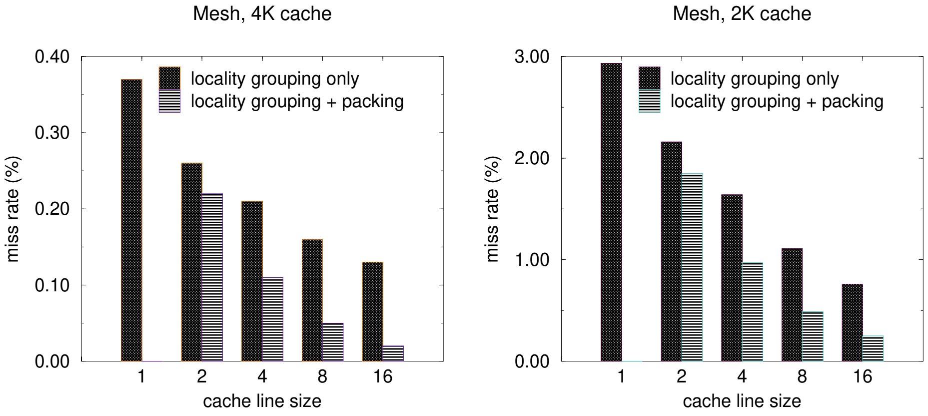


Figure 4.6 *Mesh* after locality grouping

and the combination of locality grouping and consecutive packing yields the lowest miss rate.

4.3 Compiler Support for Dynamic Data Packing

Run-time data transformations, dynamic data packing in particular, involve redirecting memory accesses to each transformed data structure. Such run-time changes complicate program transformations and induce overhead during the execution. This section first illustrates the process of data packing and the two optimizations that reduce packing overhead. More important is the compiler analysis that identifies all opportunities for data packing and its optimizations. The compiler analysis itself can guarantee correctness, but it still relies on a user to hint on profitability. The last section extends the compiler framework to collect run-time feedback and to automate the profitability analysis.

4.3.1 Packing and Packing Optimizations

The core mechanism for supporting packing is a run-time data map, which maps from the old location before data packing to the new location after data packing. Each access to a transformed array is augmented with the indirection of the corresponding run-time map. Thus the correctness of packing is ensured regardless the location and the frequency of packing. Some existing language features such as sequence and storage association in Fortran prevent a compiler from accurately detecting all accesses to a transformed array. However, this problem can be safely solved in a combination of compile, link and run-time checks described in[CCC+97].

Although the compiler support can guarantee the correctness of packing, it needs additional information to decide on the profitability of packing. Our compiler currently relies on a one-line user directive to specify whether packing should be applied, when and where packing should be carried out and which access sequence should be used to direct packing. The packing directive provides users with full power of controlling data packing, yet relieves them from any program transformation work. The last part of this section will show how the profitability analysis of packing can be automated without relying on any user-supplied directive.

A simplified dynamic program is given in Figure 4.7 to illustrate our compiler support for data packing. The kernel of *Moldyn* has two computation loops: the first loop calculates cumulative forces on each object, and the second loop calculates the new location of each object as a result of those forces. The packing directive specifies that packing is to be applied before the first loop.

```
Packing Directive: apply packing using interactions

for each pair (i,j) in interactions
   calculate_force( force[i], force[j] )
end for

for each object i
   update_location( location[i], force[i] )
end for
```

Figure 4.7 *Moldyn* kernel with a packing directive

The straightforward (unoptimized) packing produces the code shown in Figure 4.8. The call to *apply_packing* analyzes the *interactions* array, packs $force$ array and generates the run-time data map, *inter$map*. After packing, indirections are added in both loops.

The cost of data packing includes both data reorganization during packing and data redirection after packing. The first cost can be balanced by adjusting frequency of packing. Thus the cost of reorganizing data is amortized over multiple computation iterations. A compiler can make sure that this cost does not outweigh any performance

```
apply_packing( interactions[*], force[*], inter$map[*])
for each pair (i,j) in the interaction array
   calculate_force( force[ inter$map[i] ],
                    force[ inter$map[j] ] )
end for

for each object i
   update_location(location[i], force[ inter$map[i] ])
end for
```

Figure 4.8 *Moldyn* kernel after data packing

gain by either applying packing infrequently or making it adjustable at run time. As it will be shown in Chapter 6, data reorganization incurs negligible overhead in practice.

Data indirection, on the other hand, can be very expensive, because its cost is incurred on every access to a transformed array. The indirection overhead comes from two sources: the instruction overhead of indirection and the references to run-time data maps. The indirection instructions have a direct impact on the number of memory loads but the overhead becomes less significant in deeper memory hierarchy levels. However, the cost of run-time data maps has an consistent effect on all levels of cache, although this cost is likely to be small in cases where the same data map is shared by many data arrays. In addition, as shown next, the cost of indirection can be almost entirely eliminated by two compiler optimizations, *pointer update* and *array alignment*.

Pointer update modifies all references to transformed data arrays so that the indirections are no longer necessary. In the above example, this means that the references in *interactions* array are changed so that the indirections in the first loop can be completely eliminated. To implement this transformation correctly, a compiler must (1) make sure that every indirection array is associated with only one run-time data map and (2) when packing multiple times, maintain two maps for each run-time data map, one maps from the original layout and the other maps from the most recent data layout.

The indirections in the second loop can be eliminated by array alignment, which reorganizes the *location* array in the same way as the $force$ array, that is, aligns the i's element of both arrays. Two requirements are necessary for this optimization to be legal: (1) the loop iterations can be arbitrarily reordered, and (2) the range of loop iterations is identical to the range of re-mapped data. The second optimization, strictly speaking, is more than a data transformation because it reorders loop iterations. However, the reordering preserves all data dependences and therefore preserves full numerical accuracy of an application.

The example code after applying pointer update and array alignment is shown in Figure 4.9. The *update_map* array is added to map data from the last layout to the current layout. After the two transformations, all indirections through the *inter$map* array have been removed.

```
apply_packing( interactions[*], force[*],
               inter$map[*], update_map[*] )
update_indirection_array( interactions[*],
                          update_map[*] )
transform_data_array(location[*], update_map[*])

for each pair (i,j) in interactions
   calculate_force( force[i], force[j] )
end for

for each object i
   update_location( location[i], force[i] )
end for
```

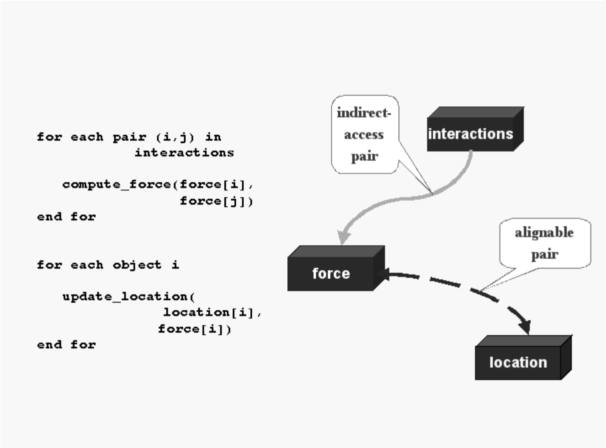
Figure 4.9 *Moldyn* kernel after packing optimizations

The overhead of array alignment can be further reduced by avoiding packing those data arrays that are not live at the point of data packing. In the above example, if the *location* array does not carry any live values at the point of packing, then the third call, which transforms *location* array, can be removed.

4.3.2 Compiler Analysis and Instrumentation

The first step of the compiler support is to find what I call *primitive packing groups*. A primitive packing group contains two sets of arrays: the set of access arrays, which hold the indirect access sequence, and the set of data arrays, which are either indirectly accessed through the first set of arrays or alignable with some arrays that are indirectly accessed.

Primitive packing groups are identified as follows. For each indirect access, the compiler finds the two arrays that are involved. They are an *indirect access pair*, and they form a primitive packing group. For each loop, if its iterations can be arbitrarily reordered, all accessed arrays are in an *alignment group*, and they form a primitive packing group where the access array set is empty. An indirect access pair and an alignment group of the *Moldyn* kernel are shown in Figure 4.10.



```
for each pair (i,j) in
            interactions
   compute_force(force[i],
                 force[j])
end for

for each object i
   update_location(
              location[i],
             force[i])
end for
```

Figure 4.10 Primitive packing groups in *Moldyn*

After finding all primitive packing groups, the compiler partitions these groups into disjoint packing partitions. Two primitive packing groups are disjoint if they do not share any array between their access array sets and between their data array sets. A union-find algorithm can efficiently perform the partitioning.

After partitioning, each disjoint packing partition is a packing candidate. The compiler then chooses those candidates that contain arrays specified in user directives. The two packing optimizations are readily applied on any packing candidate, should it become the choice of packing. Pointer update changes all arrays in the access array set; array alignment transforms all arrays in the data array set and reorders the loops that access aligned arrays.

The use of packing optimizations needs to be restricted if a packing candidate has inconsistent requirements for array alignment and pointer update. The checks for correctness are as follows. A data array can be reordered if all optimizations agree on a single layout; an access array can be updated if all its indirections point to data arrays of the same layout, or equivalently, if all its update requirement are the same. The optimizations are disabled for arrays with conflicting transformation requirements.

The correctness of array alignment requires one additional check, which is that any reordered loop must traverse the full range of all transformed arrays within the loop. If a compiler knows the exact bounds of such loops, it can restrict packing to reorder only within that range. Otherwise, a compiler must check at run time whether the range requirement is met before a loop, and if not, fall back to the unoptimized version with data indirections.

Whenever the packing optimizations are not applicable or are disabled, the compiler inserts indirections through run-time maps to all accesses to the transformed data. The overall process of compiler analysis and instrumentation is summarized in Figure 4.11.

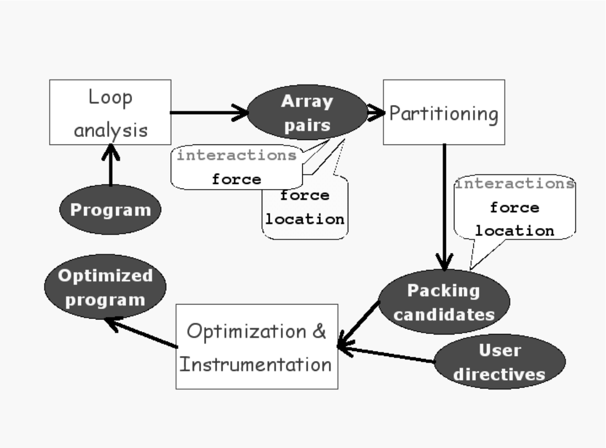


Figure 4.11 Compiler indirection analysis and packing optimization

The compiler analysis does not assume any outside knowledge of a program and its data structures. Yet it is powerful enough to identify all data indirections. In fact, the packing candidates form an indirect access graph, which identifies all data arrays as well as pointer arrays at different levels. For example, in a program with two-level indirections, an array of first-level pointers is not only an access array in one packing candidate but also a data array in another packing candidate. Both data and pointer arrays can be packed, and packing at one level is independent from all other levels.

4.3.3 Extensions to Fully Automatic Packing

Although the one-line packing directive is convenient when a user knows how to apply packing, the mandatory requirement for such a directive is not desirable in situations when a user cannot make an accurate judgement on the profitability of packing. This section discusses several straightforward extensions that can fully automate the profitability analysis, specifically, extensions that decide whether, where, and when to apply packing.

With the algorithm described in the previous section, a compiler can identify all packing candidates. For each candidate, the compiler can record the access sequence at run time and determine whether it is non-contiguous and, if so, whether packing can improve its spatial reuse. Such decisions depend on program inputs and must be made with some sort of run-time feedback system. In addition, the same data may be indirectly accessed by more than one access sequence, each may demand a different reorganization scheme. Again, run-time analysis is necessary to pick out the best packing choice.

Once the compiler chooses a packing candidate, it can place packing calls right before the place where the indirect data accesses begin. The placement requires finding the right loop level under which the whole indirect access sequence is iterated.

The frequency of packing can also be automatically determined. One efficient scheme is to monitor the average data distance in an indirect access sequence and only invoke packing routines when adjacent computations access data that are too far apart in memory. Since the overhead of data reorganization can be easily monitored at run-time, the frequency of packing can be automatically controlled to balance the cost of data reorganization.

4.4 Summary

This chapter has presented two new techniques, locality grouping and data packing, which are the run-time version of computation fusion and data grouping. Two goals have been achieved: to find the optimizations that are cost-effective at run time, and to use a compiler to automate these optimizations and to reduce their overhead.

Locality grouping brings together all the computation units involving the same data element. Its time and space cost is linear to the number of computation units. It is the least expensive among all existing reordering schemes, yet it is shown to be very powerful in a simulation study, leaving little room for additional improvement by more expensive methods. Furthermore, locality grouping is vital for the subsequent data transformation to be effective.

Data packing clusters simultaneously used data elements into adjacent memory locations. Since optimal data packing is NP-complete, the chapter presented and evaluated three heuristic-based packing algorithms and found that simple consecutive packing (at linear time and space cost) performs best when carried out after locality grouping. When evaluated on a real data set, the combined computation and data transformation reduced memory traffic by a factor of over 200.

More importantly, this chapter described a general compiler support for run-time data transformations such as data packing. The core is an analysis algorithm for detecting the structure of indirect data access in a program. This compiler analysis serves two important purposes. The first is to automatically identify all opportunities for data packing. The second is to enable two optimizations, pointer update and array alignment, which eliminate data indirections after run-time data relocation. After a new data layout is constructed by data packing, *pointer update* modifies the content of an access array to redirect it to point to the new data layout. *Array alignment* transforms other related arrays into the same data layout as the packed arrays so that the full traversal of both groups of arrays can be made contiguous.

Chapter 5

Performance Tuning and Prediction

5.1 Introduction

The preceding chapters have developed automatic compiler transformations that minimize the overall memory transfer. Although effective, automatic optimizations are not fully satisfactory for two reasons. First, a compiler may fail to optimize some part of a program because of either imprecise analysis or imperfect transformations. As compilers are taking an increasingly important role in optimizing the deep and complex memory hierarchy, their failure also becomes more dangerous and may lead to serious performance slowdown. The second limitation of compiler optimizations is their inability to provide estimation for program execution time, although such estimates would be extremely helpful in subsequent parallelization and task scheduling.

To overcome these two limitations, a compiler needs to provide support for performance tuning and prediction. The former detects and locates performance problems and therefore allows effective user tuning; the latter estimates the execution time of various program units and thus enables efficient concurrent execution.

In the past, various performance tools have been developed to support user tuning and performance prediction. However, existing tools have not been effective in practice because they either do not consider memory hierarchy or do so by pursuing the difficult measurement of memory latency. Since the exposed latency of a memory reference is determined by many factors of a machine and a program, previous tools have to rely on detailed machine simulations. Not only are simulations expensive, machine-specific and error-prone, but they also cannot predict memory hierarchy performance.

To provide a practical performance tool, this chapter investigates a bandwidth-based approach. Because memory bandwidth is the bottleneck, program performance is largely determined by its memory bandwidth utilization. On the one hand, memory bandwidth utilization determines machine utilization because the consumption of the bottleneck resource determines the consumption of the whole system. On the other hand, memory transfer time determines program execution time because the time spent on crossing the bottleneck is the time spent on crossing the whole system. Therefore, we can monitor program performance based on its memory bandwidth utilization, and we can predict its performance based on its memory transfer time. Based on these two observations, the following sections present the design of a bandwidth-based performance tool, describe its use in performance tuning and predictions, and discuss several extensions for more accurate analysis.

5.2 Bandwidth-based Performance Tool

The bandwidth-based performance tool takes as input, a source program along with its inputs and parameters for the target machine. It first estimates the total amount of data transfer between memory and cache. This figure is then used to either predict the performance without running the program or locate memory hierarchy performance problems given the actual running time. Figure 5.1 shows the structure of the tool, as well as its inputs and outputs.

5.2.1 Data Analysis

The core support of the tool is the data analysis that estimates the total amount of data transfer between memory and cache. First, a compiler partitions the program into a hierarchy of computation units. A computation unit is defined as a segment of the program that accesses data larger than cache. Given a loop structure, a compiler can automatically measure the bounds and stride of array access through, for example, interprocedural bounded-section analysis developed by Havlak and Kennedy[HK91]. The bounded array sections is then used to calculate the total amount of data access and to determine whether the amount is greater than the size of the cache. The additional amount of memory transfer due to cache interference can be approximated by the technique given by Ferrante et al[FST91]. Once a program is partitioned into a hierarchy of computation units, a bottom-up pass is needed to summarize the total amount of memory access for each computation unit in the program hierarchy until the root node—the whole program.

Since exact data analysis requires precise information on the bounds of loops and coefficients of array access, the analysis step needs to have run-time program input to make the correct estimation, especially for programs with varying input sizes. In certain cases, however, the number of iterations is still unknown until the end of

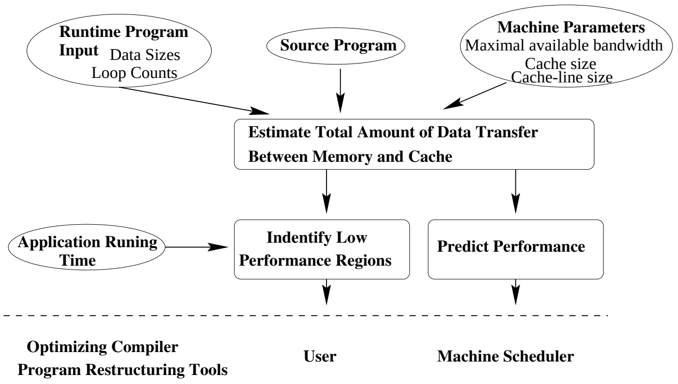


Figure 5.1 Structure of the performance tool

execution. An example is an iterative refinement computation, whose termination point is determined by a convergence test at run time. In these cases, the analysis can represent the total amount of memory access as a symbolic formula with the number of iterations as an unknown term. A compiler can still successfully identify the amount of data access within each iteration and provide performance tuning and prediction at the granularity of one computation iteration.

5.2.2 Integration with Compiler

Since all data-analysis steps are performed statically, the performance tool can be integrated into the program compiler. In fact, an optimizing compiler may already have these analyses built in. So including this tool into the compiler is not only feasible but also straightforward. Although the tool requires additional information about the run-time program inputs, the data analysis can proceed at compile time with symbolic program inputs and then re-evaluate the symbolic results before execution.

The integration of the tool into the compiler is not only feasible but also profitable for both the tool and the compiler. First, the tool should be aware of certain compiler transformations such as data-reuse optimizations because they may change the actual amount of memory transfer. The most notable is global fusion and data grouping, presented in previous chapters, which can radically change the structure of both the computation and data of a program and can reduce the overall amount of memory transfer by integral factors. The performance tool must know these high-level transformations for it to obtain an accurate estimate of memory transfer.

In addition to helping data analysis, the integration of the tool helps the compiler to make better optimization decisions. Since the tool has the additional knowledge of the program inputs, it can supply this information into the compiler. The precise knowledge of run-time data and machine parameters is often necessary to certain compiler optimizations such as cache blocking and array padding. Therefore, the integration of the compiler and the performance tool improves not only the accuracy of the performance tool but also the effectiveness of the compiler.

5.3 Performance Tuning and Prediction

Performance Tuning

In bandwidth-based performance tuning, a compiler searches for computation units that have abnormally low memory bandwidth utilization. Because of memory bandwidth bottleneck, a low bandwidth utilization implies a low utilization of all other hardware resources, therefore signaling an opportunity for tuning. A compiler can automatically identify all such tuning opportunities in the following two steps.

1. The first step executes the program and collects the running time of all its computation units. The achieved memory bandwidth is calculated by dividing the data transfer of each computation unit with its execution time. The achieved memory bandwidth is then compared with machine memory bandwidth to obtain the bandwidth utilization.
2. Second, the tool singles out the computation units that have low memory bandwidth utilization as candidates for performance tuning. For each candidate, the tool calculates the potential performance gain as the difference between the current execution time and the predicted execution time assuming full bandwidth utilization. The tuning candidates are ordered by their potential performance gain and then presented to a user.

Bandwidth-based performance tuning requires no special hardware support or software simulation. It is well suited for different machines and compilers because the use of actual running time includes the effect of all levels of compiler and hardware optimizations. Therefore, it is not only automatic, but also accurate and widely applicable.

Bandwidth-based performance tuning does not necessarily rely on compiler-directed data analysis when applied on machines with hardware counters such as MIPS R10K and Intel Pentium III. The hardware counters can accurately measure the number and the size of memory transfers. With these counters, bandwidth-based tuning can be applied to programs that are not amenable to static compiler analysis. However, compiler analysis should be used whenever feasible for three reasons. First, source-level analysis is necessary to partition a program into computation units and help a user to understand the performance at the source level. Second, static analysis is more accurate for tuning because it can identify the problem of excessive conflict misses, while hardware counters cannot distinguish different types of misses. Third, the compiler-directed analysis is more portable because it can be applied to all machine architectures including those with no hardware counters.

Performance Prediction

When a program uses all or most of machine memory bandwidth, its execution time can be predicted by its estimated memory-transfer time, that is, by dividing the total amount of memory transfer with the available memory bandwidth. This bandwidth-based prediction is simple, accurate and widely applicable to different machines, applications and parallelization schemes.

The assumption that a program utilizes all available bandwidth is not always true—some parts of the program may have a low memory throughput even after performance tuning. However, low memory throughput should happen very infrequently and it should not seriously distort the overall memory bandwidth utilization. The variations in the overall utilization should not introduce large errors into performance prediction. Otherwise, the program must have a performance bottleneck other than memory bandwidth. The next section discusses techniques for detecting other resource bottlenecks such as loop recurrence or bandwidth saturation between caches.

5.4 Extensions to More Accurate Estimation

Although the latency of arithmetic operations on modern machines is very small compared to the time of memory transfer, it is still possible that computations in a loop recurrence may involve so many operations that they become the performance bottleneck. So the tool should identify such cases with the computation-interlock analysis developed by Callahan et al[CCK88].

Excessive misses at other levels of memory hierarchy can be more expensive than memory transfer. The examples are excessive register loads/stores, higher-level cache misses, and TLB misses. To correctly detect these cases, the performance tool needs to measure the resource consumption on other levels of memory hierarchy. In fact, the tool can extend its data analysis to measure the number of higher-level cache misses and TLB misses, which are in fact special cases of the existing data analysis.

On a machine with distributed memory modules, memory references may incur remote data access. When a remote access is bandwidth limited, the tool can estimate its access time with the same bandwidth-based method except that it needs to consider the communication bandwidth in addition to memory bandwidth. The bandwidth-based method also needs to model bandwidth contention either at a memory module or in the network. When a remote access is not bandwidth-constrained, we can train the performance estimator to recognize cases of long memory latency using the idea of training sets[BFKK91]. The bandwidth-based tuning tool can automatically collect such cases from applications because they do not fully utilize bandwidth.

Coherence misses in parallel programs should also be measured if they carry a significant cost. A compiler can detect coherence misses, especially for compiler parallelized code[McI97].

5.5 Summary

This chapter has presented a design of a bandwidth-based tool for performance tuning and prediction. The central part of the tool is the compiler analysis that divides a program into computation phases and estimates the total amount of memory transfer for each computation phase on a given machine. For performance tuning, the tool uses the data estimation and program execution time to calculate memory bandwidth utilization. Then it picks out those computation phases with low memory bandwidth utilization for further tuning. For performance prediction, the tool approximates program execution time by dividing the total amount of memory transfer with machine memory bandwidth. To improve the accuracy of bandwidth-based method and to consider cases where memory bandwidth is not the critical resource, the proposed tool is augmented with additional compiler analysis to monitor the effect of latency and bandwidth constraint in other parts of a computer system.

The bandwidth-based approach promises to be much more efficient, yet more accurate than previous methods that are based on monitoring memory latency. Instead of simulating individual memory access, the new tool assesses the cost of all memory references, that is, the time of all memory transfer. The new tool is also much simpler because it focuses on only very large data structures. By changing from latency based to bandwidth based, the tool avoids the cost of previous techniques and enables fast, accurate, and portable performance tuning and prediction.

Chapter 6

Evaluation

This chapter evaluates the compiler strategy of global and dynamic computation fusion and data grouping. Section 1 describes the compiler implementation. Section 2 explains the experimental setup, which measures two classes of applications: regular programs with structured loops and predictable data access, and irregular and dynamic programs with unstructured computation and unpredictable data access. Section 3 and Section 4 describe the benchmark applications, the applied transformations and the experiment results for each class of applications. Section 5 evaluates the bandwidth-based performance tool. Finally, Section 6 summarizes the findings.

6.1 Implementation

The implementation is based on the D Compiler System at Rice University. The compiler performs whole program compilation given all source files of an input program. It uses a powerful value-numbering package to handle symbolic variables and expressions inside each subroutine and parameter passing between subroutines. It has a standard set of loop and dependence analysis, data flow analysis and interprocedural analysis. The D Compiler compiles programs written in Fortran 77 and consequently it does not handle recursion. However, recursion should present no fundamental obstacles to the new methods described in this dissertation.

This research have implemented loop fusion, array regrouping and dynamic data packing for experimental evaluation. The following sections describe the implementation of these three new techniques.

6.1.1 Maximal Loop Fusion

For each loop, the compiler summarizes its data access by its data footprint. For each dimension of an array, a data footprint describes whether the loop accesses the whole dimension, a number of elements on the border, or a loop-variant section (a range enclosing the loop index variable). Data dependence is tested by the intersection of footprints. The range information is also used to calculate the minimal alignment factor between loops.

Loop fusion is carried out by applying the fusion algorithm given in Figure 2.7 level by level from the outermost to the innermost. The current implementation calculates data footprints, aligns loops and schedules non-loop statements. Iteration reordering is not yet implemented but the compiler signals the places where it is needed. Only one program, *Swim*, required splitting, which was done by hand.

For multi-level loops, loop fusion orders loop levels to maximize the benefit of fusion, as specified by the algorithm in Figure 2.8. The first loop level to fuse is the one that produces the fewest loop nests after fusion. In the experiment, however, this was largely unnecessary, as computations were mostly symmetric. One exception in our test cases was *Tomcatv*, where level ordering (loop interchange) was performed by hand.

Code generation is straightforward as mappings from the old iteration space to the fused iteration space. Currently, the code is generated by the Omega library[Pug92], which has been integrated into the D compiler system. Omega worked well for small programs, where the compilation time was under one minute for all kernels. For the full application *SP*, however, code generation took four minutes for one-level fusion but one hour and a half for three-level fusion. In contrast, the fusion analysis took about two minutes for one-level fusion and four minutes for three-level fusion. A direct code generation scheme has been designed; its cost is linear to the number of loop levels. But the implementation is currently not available.

6.1.2 Inter-array Data Regrouping

The analysis for data regrouping is trivial with data footprints. After fusion, data regrouping is applied level by level on fused loops as specified by the algorithm in Figure 3.4. However, the implementation makes two modifications. First, SGI compiler does a poor job when arrays are fully interleaved at the innermost data dimension. So the compiler instead groups arrays up to the second innermost dimension. This restriction may result in grouping in the less desired dimension, as in the case of *Tomcatv*. The other restriction is due to the limitation of Fortran language, which does not allow non-uniform array dimensions. In cases where multi-level regrouping produced non-uniform arrays, manual changes were made to not to group at outer data dimensions.

Code generation for array regrouping is semi-automatic. The compiler generates the choice of regrouping. Then new array declarations are added by hand, and array references are transformed through the macro processor *cpp*. This scheme worked well when the name of arrays was consistent and unique throughout the program, which was the case for most of the programs tested. Manual changes were made to *Magi* to make one data array global instead of passing by parameters.

Data regrouping was performed by hand for irregular and dynamic applications because the experiment was performed before the implementation became available. Since these applications used one-dimensional arrays and performed indirected access, one-dimensional grouping is sufficient. Therefore, the manual approximation of data regrouping was trivial.

6.1.3 Data Packing and Its Optimizations

The implementation for data packing and its optimization follows the algorithm described in Section 4.3.2 and the steps illustrated in Figure 4.11. The compiler recognizes the structure of indirect access in a program and identifies all packing opportunities. A one-line user directive is used to specify which array should be packed, as well as where and how often it should be packed. The two packing optimizations, pointer update and array alignment, are applied automatically.

The current implementation does not work on programs where the indirect access sequence is incrementally computed because the one-line directive requires the existence of a full access sequence. A possible extension would be to allow user to specify a region of computation in which to apply packing so that the compiler can record the full access sequence at run time. The other restriction of the current implementation is due to conservative handling of array parameter passing. For each subroutine with array parameters, the implementation does not allow two different array layouts to be passed to the same formal parameter. This problem can be solved by propagating array layout information in a way similar to interprocedural constant propagation or data type analysis and then cloning the subroutine for each reachable array layout. In the programs encountered, however, there is no need for such cloning.

6.2 Experimental Design

All programs are measured on one of the three SGI machines: SGI Origin2000 with R10K processors, SGI Origin2000 with R12K processors, and SGI O2 with a R10K processor. Both R12K and R10K provide hardware counters that measure cache misses and other hardware events with high accuracy. All machines have two caches: L1 is 32KB in size and uses 32-byte cache lines, L2 uses 128-byte cache lines, and the size of L2 is 1MB for O2 and 4MB for Origin2000 (with either R10K or R12K). Both caches are two-way set associative. All processors achieve good latency hiding as a result of dynamic, out-of-order instruction issuing and compiler-directed prefetching. All applications are compiled with the highest optimization flag and prefetching turned on (f77 -n32 -mips4 -Ofast)[6]. The SGI compiler is MIPSpro Version 7.30.

The experiment on irregular and dynamic applications used the slower R10K processor on Origin2000 because the newer machine with R12K was not available at the time of the experiment; the same is true for the evaluation of performance tuning and prediction. The later evaluation of regular programs used Origin2000 with R12K processors. In addition, it used SGI O2 for a direct comparison with an earlier work by another group.

All transformations preserve dependences except *locality grouping*, which is applied only once to one application. All optimized programs produce the identical output as their unoptimized version.

The effect of optimizations is measured on execution time and the number of L1, L2 and TLB misses. Cache misses represent only memory read traffic; therefore, they are not equal to the amount of bandwidth consumption. However, the improved cache reuse has similar effects on memory reads as it does on memory writebacks. Data regrouping should eliminate unnecessary writebacks, but the function is disabled because it does not regroup at the innermost data dimension. Since none of the evaluated techniques optimizes specially for memory writebacks, they are not measured in this experiment. Nevertheless, all the executables of the original and the optimized programs have been preserved, and any additional measurement can be made in the future if necessary.

6.3 Effect on Regular Applications

This section evaluates the global strategy of maximal loop fusion as the first step and inter-array data regrouping as the second. The programs are written in loop and array structures with a data access pattern that is mostly compile-time predictable.

---

[6] The only exception is *Magi*, where user specified (-mp -64 -r10000 -O3 -SWP:=ON -mips4 -OPT:IEEE_arithmetic=3:round off=3:alias=restrict).

Irregular and dynamic applications are measured in the next section, where dynamic optimizations are tested.

6.3.1 Applications

Loop fusion and data regrouping are tested on four applications described in Figure 6.1. The applications come from SPEC and NAS benchmark suite except *ADI*, which is a self-written kernel with separate loops processing boundary conditions. Since all programs use iterative algorithms, only the loops inside the time-step loop are counted.

name	source	input size	No. lines	loops (levels)	arrays
Swim	SPEC95	513x513	429	6 (1-2)	15
Tomcatv	SPEC95	513x513	221	5 (1-2)	7
ADI	self-written	2Kx2K	108	6 (1-2)	3
SP	NAS/NPB Serial v2.3	class B, 3 iterations	1141	67 (2-4)	15

Figure 6.1 Descriptions of Regular applications

6.3.2 Transformations Applied

Loop fusion and data regrouping have been applied to all programs. In addition, an input program is processed by four preliminary transformations before applying loop fusion. The first is procedure inlining, which brings all computation loops into a single procedure. The next is array splitting and loop unrolling, which eliminates data dimensions of a small constant size and loops that iterate those dimensions. The third step is loop distribution. Finally, the last step propagates constants into loop statements. The compiler performs loop unrolling and constant propagation automatically. Currently, array splitting requires a user to specify the names, and inlining is done by hand. However, both transformations can be automated with additional implementation.

6.3.3 Effect of Transformations

The effect of optimizations is shown in Figure 6.2. All results are collected on Origin2000 with R12K processors except *Swim*, which is on SGI O2. The graphs for the first three applications show four sets of bars: the original performance (normalized to 1), the effect of only loop fusion, the effect of only data grouping, and the effect of loop fusion plus data regrouping. For *SP*, one additional set of bars is used to show the effect of fusing one loop level instead of fusing all loop levels. The execution time and original miss rates are also given in the figures; however, reductions are measured on the number of misses, not on the miss rate.

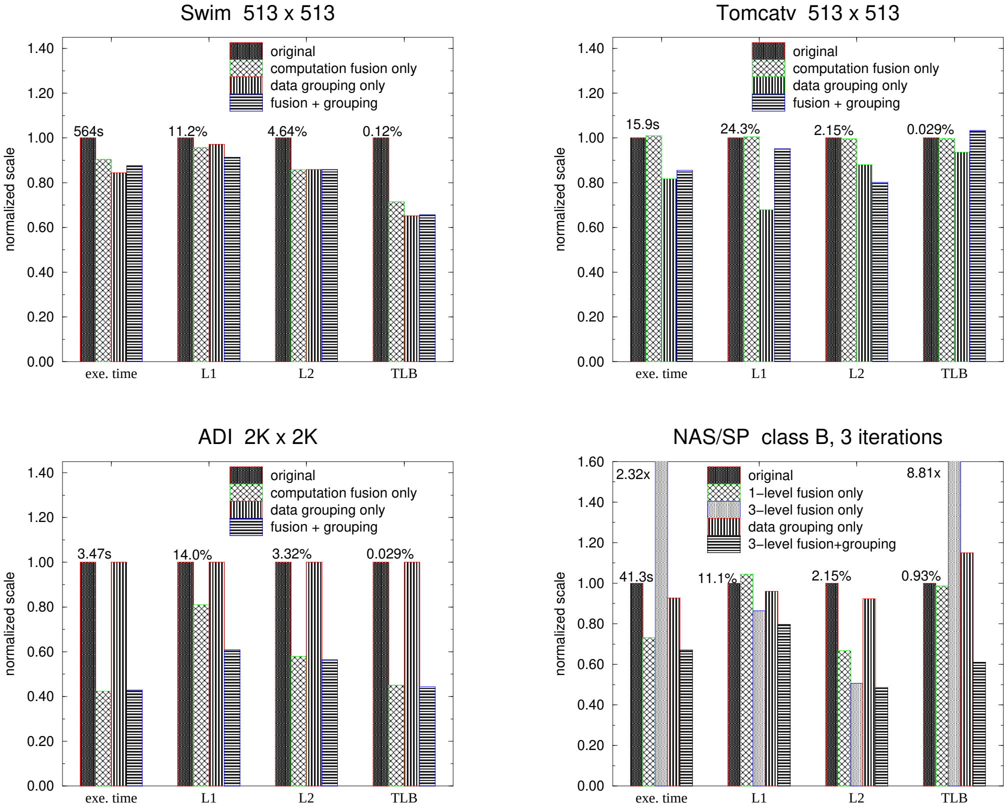


Figure 6.2 Effect of transformations on regular applications

The performance of *Swim* is reported for SGI O2 because it has the same cache configuration as SGI Octane, a machine used in the work of iteration slicing by Pugh and Rosser[PR99]. Maximal loop fusion fuses all loop nests with the help of loop splitting and achieves the same improvement (10%) as Pugh and Rosser reported for iteration slicing. The succeeding data grouping, shown by the fourth bar in each cluster, merges 13 arrays into 3 and shortens execution time by 2% more because of the additional reduction on L1 and TLB misses.

Compared to data regrouping with loop fusion, data regrouping without loop fusion (shown by the third bar of each cluster) merges 13 arrays into 5 and results in similar reductions on L2 and TLB but causes 6% more L1 misses. Data regrouping without loop fusion achieves the shortest execution time, a reduction of 16%. The reason it performs better than the combined strategy is that the benefit of loop fusion does not outweigh its instruction overhead when the data size is as small as in this program. When the input size is large, the benefit of loop fusion should become dominant. We will use much larger programs for the last two test cases.

One side effect of data regrouping seen by this program is 61% increase in the number of graduated register loads. The overhead shows that source-level data regrouping confuses the back-end compiler and results in less effective register allocation. However, the problem should disappear if data regrouping is applied by the back-end compiler itself. Among all test cases, *Swim* is the only one where data regrouping causes an increased amount of register traffic.

Compared to SGI O2, running *Swim* on Origin2000 incurs 66% fewer L2 misses because of the larger L2 cache on Origin2000. Loop fusion decreases performance by 6% on Origin2000, but the loss is recovered by data regrouping. Data regrouping without loop fusion reduces cache and TLB misses but increases register loads by 35%. Since cache is not the bottleneck for this data size on Origin2000, the increase in register traffic has a dramatic effect of increasing execution time by 16%. However, as explained before, data regrouping will not suffer from this overhead if it is applied by the back-end compiler.

Tomcatv has two pipelined computations progressing along reverse directions, so multi-level loop fusion permutes non-conflicting loops to the outside to enable fusion at the outer loop level. Data regrouping merges 7 arrays into 4. As before, the arrays are grouped at the outer data dimension instead of inner data dimension in order to avoid the poor code generation of the SGI compiler (see Section 6.1.2). Figure 6.2 shows the program with an input size of 513x513. Loop fusion alone decreases performance by 1%, but the combined transformation reduces L1 misses by 5%, L2 misses by 20% and overall execution time by 16%. Data regrouping after loop fusion increases TLB misses by 3% because of the side effect of grouping at the outer data dimension. The side effect is small and becomes visible only because of the extremely low TLB miss rate of this program, which is 0.03%.

Data regrouping without loop fusion groups 7 arrays into 5 and reduces L1 misses by 32%, L2 misses by 12%, TLB misses by 6% and as a result reduces execution time by 18%. These reductions are larger than those of regrouping with loop fusion except for L2 misses, where the combined transformation eliminates 8% more misses with the help of loop fusion. Although the regrouping is similar with and without loop fusion, the effect on cache and TLB misses differs significantly for two reasons. The first is that the input data size is small, and the overhead of loop fusion is pronounced. The other reason is that after loop interchange, the inner loops iterate through the outer data dimension, making memory performance sensitive to small changes in data layout. For example, regrouping with fusion increases TLB misses by 3%, but regrouping without fusion decreases TLB misses by 6%. These variations should not occur if arrays are regrouped at the inner data dimension by the back-end compiler. In addition, data regrouping can also permute data dimensions. However, determining the best order for data dimensions is a NP-complete problem. The current algorithm is conservative and does not allow transformations like this because they may be detrimental to overall performance.

The original data input is 257x257 for *Tomcatv*. At this small size on Origin2000, *Tomcatv* exhibits similar behavior as *Swim*: loop fusion decreases performance by 2% but data regrouping recovers the loss and improves performance by 1%. On SGI O2, loop fusion decreases performance by 1%, but the combined transformation improves performance by 5%.

ADI uses the largest input size and consequently enjoys the highest improvement. The reduction is 39% for L1 misses, 44% for L2 and 56% for TLB. The execution time is reduced by 57%, a speedup of 2.33. Since only three arrays are used in the program, data regrouping has little benefit on L2, TLB and the execution time, but it reduces L1 misses by 20%. Without loop fusion, however, data regrouping sees no chance in merging any array. Therefore, regrouping without loop fusion has no effect on performance.

Program changes for *SP* SP is a full application and deserves special attention in evaluating the global strategy. The main computation subroutine, *adi*, uses 15 global data arrays in 218 loops, organized in 67 nests (after inlining). Loop distribution and loop unrolling result in 482 loops at three levels—157 loops at the first level, 161 at the second, and 164 at the third. One-level loop fusion merges 157 outer-most loops into 8 loop nests. The performance is shown by the second bar in the lower-right graph of Figure 6.2. The full fusion further fuses loops in the remaining two levels and produces 13 loops at the second level and 17 at the third. The performance of full fusion is shown by the third bar in the graph.

The fourth and fifth sets of bars show the effect of data regrouping with and without loop fusion. The original program has 15 global arrays. Array splitting resulted in 42 arrays. After full loop fusion, data regrouping combines 42 arrays into 17 new ones. The choice of regrouping is very different from the specification given by the programmer. For example, the third new array consists of four original arrays: $\{ainv(N, N, N), us(N, N, N), qs(N, N, N), u(N, N, N, 1-5)\}$, and the 15th new array includes two disjoint sections of an original array: $\{lhs(N, N, N, 6-8), lhs(N, N, N, 11-13)\}$.

One-level fusion increases L1 misses by 5%, but reduces L2 misses by 33% and execution time by 27%, signaling that the original performance bottleneck was on memory bandwidth. Fusing all levels eliminates half of the L2 misses (49%). However, it creates too much data access in the innermost loop and causes 8 times more TLB misses. The performance is slowed by a factor of 2.32. Data regrouping, however, merges related data in contiguous memory and achieves the best performance. It reduces L1 misses by 20%, L2 by 51% and TLB by 39%. The execution time is shortened by one third (33%), a speedup of 1.5 (from 64.5 Mf/s to 96.2 Mf/s).

Without loop fusion, however, data regrouping can merge only two original arrays, $lhs(N, N, N, 7)$ and $lhs(N, N, N, 8)$. Consequently, data regrouping without loop fusion obtains only a modest improvement. It reduces L1 misses by 4%, L2 misses by 8%, and execution time by 7%. It actually increases TLB misses by 15%.

Recall that the motivation for maximal loop fusion comes from the simulation study on reuse distances in Section 2.2, where reuse-driven execution on a perfect machine found a large benefit from fusing computations on the same data. The purpose of loop fusion, then, is to realize this benefit on a real machine. Figure 6.3 compares the effect of loop fusion with that of reuse-driven execution on *SP*. It shows the reuse-distance curve of three versions of *SP*: the original program order, reuse-driven execution order, and the transformed program order after maximal loop fusion. Maximal loop fusion reduces 45% evadable reuses, which is not as good as the 63% reduction by the ideal reuse-driven execution. However, maximal fusion does realize a fairly large portion of its potential. Furthermore, the reduction on evadable reuses is very close to the reduction of L2 misses on Origin2000 (51%), indicating that the measurement of reuse distance closely matches L2 cache performance on a real machine.

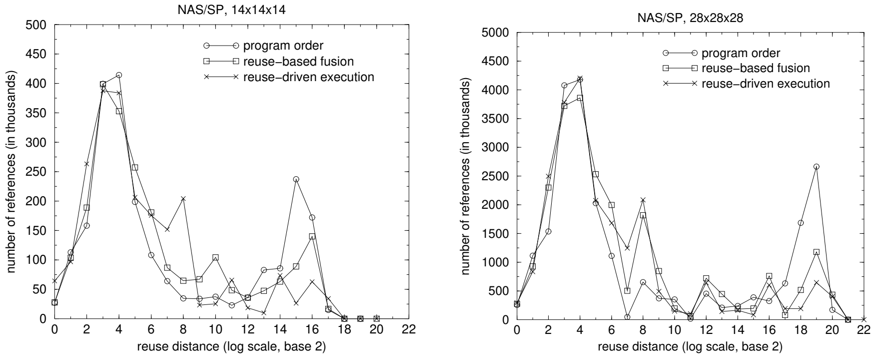


Figure 6.3 Reuse distances of *NAS/SP* after maximal fusion

The above evaluation has verified the effectiveness of the global optimization strategy. Maximal loop fusion realizes much of the potential benefit of computation fusion and brings together data reuses among all parts of a program. Data regrouping eliminates the overhead of loop fusion and translates the reduction on memory traffic to the reduction on execution time. Together, these two techniques significantly improve global cache reuse for the whole program.

6.4 Effect on Irregular and Dynamic Applications

In irregular and dynamic applications, the content of and access order to certain data structures are unknown until run time and may change during the execution. This section evaluates the effectiveness of dynamic optimizations, developed in Chapter 4. The global optimization, data regrouping, is also applied.

6.4.1 Applications

Figure 6.4 lists the four irregular and dynamic applications used in the evaluation, along with their description, programming language and code size. Three scientific simulation applications from molecular dynamics, structural mechanics and hydrodynamics are used. Despite the difference in their physical model and computation, they have similar dynamic data access patterns in which objects interact with their neighbors. *Moldyn* and *mesh* are well-known benchmarks. A large input data set is used for *moldyn* with random initialization. *Mesh* has a user-supplied input set. *Magi* is a full, real-world application consisting of almost 10,000 lines of Fortran code.

In addition to the three simulation programs, a sparse-matrix benchmark is included to show the effect of packing on irregular data accesses in such applications.

name	description	source	language	lines
Moldyn	molecule dynamics simulation	Chaos group	f77	660
Mesh	structural simulation	Chaos group	C	932
Magi	particle hydrodynamics	DoD	f90/f77	9339
NAS-CG	sparse matrix-vector multiplication	NAS/NPB Serial v2.3	f77	1141

Figure 6.4 Descriptions of irregular and dynamic applications

application	input size	source of input	exe. time
Moldyn	256K particles, 27.4M interactions, 1 iteration	random initialization	53.2 sec
Mesh	9.4K nodes, 60K edges, 20 iterations	Chaos group	8.14 sec
Magi	28K particles, 253 cycles	DoD	885 sec
NAS-CG	14K non-zero entries, 15 iterations	NASA/NPB Serial 2.3, Class A	48.3 sec

Figure 6.5 Input sizes of irregular and dynamic applications

Figure 6.5 gives the input size for each application, the sources of the data inputs, and the execution time before applying optimizations. The working set is significantly larger than the L1 cache for all applications. $Mesh$, $Magi$ and $NAS-CG$ are a little bit larger than L2. $Moldyn$ has the largest data input and its data size is significantly greater than the size of L2.

6.4.2 Transformations Applied

The transformations were applied in the following order: locality grouping, data regrouping, dynamic data packing and packing optimizations. Since the access sequence is already transformed by locality grouping, consecutive packing is used for all cases because of the observation made in Section 4.2.2. (One test case, $NAS-CG$, accesses each element only once, therefore consecutive packing is optimal.)

Figure 6.6 lists, for each application, the optimizations applied and the program components measured. Each of the base programs came with one or more of the three optimizations done by hand. Such cases are labeled with a '+' sign in the table. The 'V' signs indicate the optimizations added, except in the case of NAS-CG. The base program of $NAS-CG$ came with data packing already done by hand, but I removed it for the purpose of demonstrating the effect of packing. I do not consider hand-applied packing practical because of the complexity of transforming tens of arrays repeatedly at run-time for a large program.

application	optimizations applied			program components measured
	locality grouping	regrouping	packing	
Moldyn	+	V	V	function `Compute_Force()`
Mesh	V	no effect	V	full application
Magi	+	V	V	full application
NAS-CG	n/a	no effect	V/+	full application

Figure 6.6 Transformations applied to irregular and dynamic applications

Locality grouping and data regrouping were inserted by hand. Data regrouping was applied to all programs but found opportunities only for *Moldyn* and *Magi*. Data packing of *moldyn* and *CG* was performed automatically by our compiler given a one-line directive of packing. The same compiler packing algorithm was applied to *mesh* by hand because our compiler infrastructure cannot yet compile C. Unlike other programs, $Magi$ is written in Fortran90 and computes the interaction list incrementally. I slightly modified the source to let it run through the Fortran77 front-end and inserted a loop to collect the overall data access sequence. Then our compiler successfully applied base packing transformation on the whole program. The application of the two compiler optimizations were semi-automatic: I inserted a 3-line loop to perform pointer update; and I annotated a few dependence-free loops which otherwise would not be recognized by the compiler due to the presence of procedural calls inside them. All other transformations are performed by the compiler. The optimized packing reorganizes a total of 45 arrays in *magi*.

The original program is referred to as the base program and the transformed version as the optimized program. For NAS-CG, the base program refers to the version with no packing. Dynamic data packing is applied only once in each application except *magi* where data are repacked every 75 iterations.

6.4.3 Effect of Transformations

The four graphs of Figure 6.7 show the effect of the three transformations. The first plots the effect of optimizations on the execution speed. The first bar of each application is the normalized performance (normalized to 1) of the base version. The other bars show the performance after applying each transformation. Since not all transformations are necessary, an application may not have all three bars. The second bar, if shown, shows the speedup of locality grouping. The third and fourth bars show the speedup due to data regrouping and data packing. The other three graphs are organized in the same way, except that they are showing the reduction on the number of L1, L2 and TLB misses. The graphs include the miss rate of the base program, but the reduction is on the total number of misses, not on the miss rate.

Effect of Locality Grouping and Data Regrouping

Locality grouping eliminates over half of L1 and L2 misses in *mesh* and improves performance by 20%. In addition, locality grouping avails the program for data packing, which further reduces L1 misses by 35%. Without the locality-grouping step, however, consecutive packing not only results in no improvement but also incurs 5% more L1 misses and 54% more L2 misses. This confirms the observation from our simulation study that locality grouping is critical for the later data optimization to be effective.

Data regrouping significantly improves *moldyn* and *magi*. *Magi* has multiple computation phases, data regrouping splits 22 arrays into 26 and regroups them into 6 new arrays. As a result of better spatial reuse, the execution time is improved by a factor of 1.32 and cache misses are reduced by 38% for L1, 17% for L2, and 47% for TLB. By contrast, merging all 26 arrays improves performance by only 12%, reduces L1 misses by 35%, and as a side effect, increases L2 misses by 32%. Data regrouping is even more effective on *moldyn*, eliminating 70% of L1 and L2 misses and almost doubling the execution speed.

Effect of Dynamic Data Packing

Data packing is applied to all four applications after locality grouping and data regrouping. It further improves performance in all cases. For *moldyn*, packing improves performance by a factor of 1.6 and reduces L2 misses by 21% and TLB misses by 88%

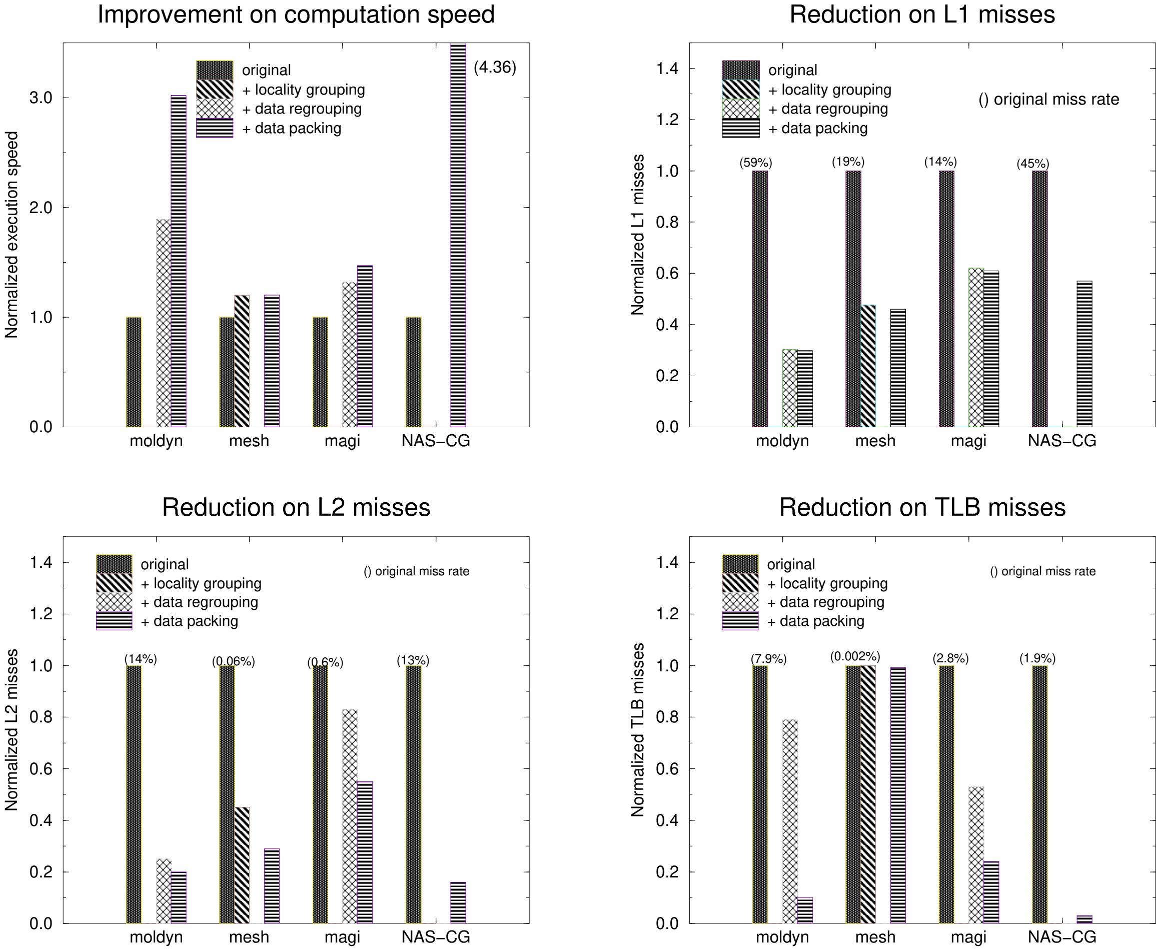


Figure 6.7 Effect of transformations on irregular and dynamic applications

over the version after data regrouping. For $NAS - CG$, the speedup is 4.36 and the amount of reduction is 44% for L1, 85% for L2 and over 97% for TLB.

For *mesh* after locality grouping, packing slightly improves performance and reduces misses by additional 3% for L1 and 35% for L2. The main reason for the modest improvement on L1 is that the data granularity (24 bytes) is close to the size of L1 cache lines (32 bytes), leaving little room for additional spatial reuse. In addition, packing is directed by the traversal of edges, which does not work as well during the traversal of faces. The number of L1 misses is reduced by over 6% during edge traversals, but the reduction is less than 1% during face traversals. Since the input data set almost fits in L2, the significant reduction in L2 misses does not produce a visible effect on the execution time.

When applied after data regrouping on *magi*, packing speeds up the computation by another 70 seconds (12%) and reduces L1 misses by 33% and TLB misses by 55%. Because of the relatively small input data set, L2 and TLB misses are not a dominant factor in performance. As a result, the speed improvement is not as pronounced as the reduction in these misses.

Overall, packing achieves a significant reduction in the number of cache misses especially for L2 and TLB, where opportunities for spatial reuse are abundant. The reduction in L2 misses ranges from 21% to 84% for all four applications; the reduction in TLB misses ranges from 55% to 97% except for *mesh*, whose working set fits in TLB.

Packing Overhead and the Effect of Compiler Optimizations

The cost of dynamic data packing comes from the overhead of data reorganization and the cost of indirect memory accesses. The time spent in packing has a negligible effect on performance in all three applications measured. Packing time is 13% of the time of one computation iteration in *moldyn*, and 5.4% in *mesh*. When packing is applied for every 20 iterations, the cost is less than 0.7% in *moldyn* and 0.3% in *mesh*. *Magi* packs data every 75 iterations and spends less than 0.15% of time on packing routines.

The cost of data indirection after packing can be mostly eliminated by two compiler optimizations described in Section 4.3.1. Figure 6.8 shows the effect of these two compiler optimizations on all four applications tested.

The upper-left graph shows that, for *moldyn*, the indirections (that can be optimized away) account for 10% of memory loads, 22% of L1 misses, 19% of L2 misses and 37% of TLB misses. After the elimination of the indirections and the references to the run-time map, execution time was reduced by 27%, a speedup of 1.37. The improvement in *mesh* is even larger. In this case, the indirections account for 87% of the loads from memory, in part because *mesh* is written in C and the compiler does not do a good job of optimizing array references. Since the excessive number of memory loads dominates execution time, the compiler optimizations achieve a similar reduction (82%) in execution time. The number of loads is increased in *magi* after the optimizations because array alignment transforms 19 more arrays than the

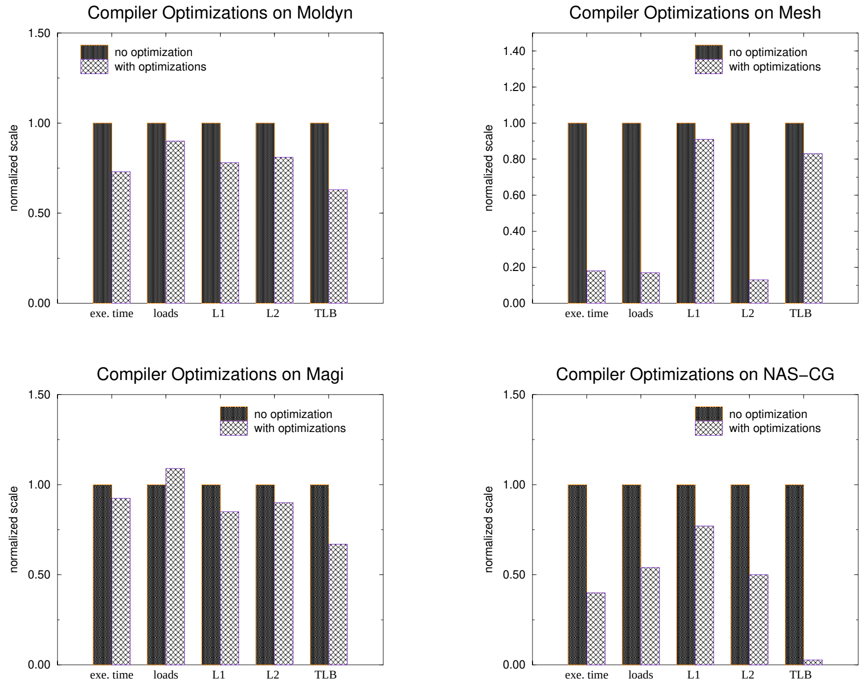


Figure 6.8 Effect of compiler optimizations for data packing

base packing, and not all indirections to these arrays can be eliminated. Despite the increased number of memory loads, the cache misses and TLB misses are reduced by 10% to 33%, and the overall speed is improved by 8%. For $NAS - CG$, the compiler recognizes that matrix entries are accessed in stride-one fashion and consequently, the compiler replaces the indirection accesses with direct stride-one iteration of the reorganized data array. The transformed matrix-vector multiply kernel has the equally efficient data access as the original hand-coded version. As a result, the number of loads and cache misses is reduced by 23% to 50%. The TLB working set fits in machine's TLB buffer after the optimizations, removing 97% of TLB misses. The execution time is reduced by 60%, a speedup of 2.47.

6.5 Effect of Performance Tuning and Predication

This section evaluates bandwidth-based performance tuning and prediction on a well-known benchmark application, *SP* from NASA, on SGI Origin2000 with R10K processors. *NAS/SP* is a complete application with over 20 subroutines and 3000 lines of Fortran77 code. Since the program consists of sequences of regular loop nests, it is partitioned into two levels of computation phases—subroutines and then loop nests. Class-B input is used and only three iterations are ran to save the experiment time.

Since the implementation of compiler analysis was not complete at the time of experiment, the following evaluation does not have compiler estimation of memory transfer. Instead, it used manual approximation and machine hardware counters.

Performance Tuning

Tuning opportunities are identified by measuring the bandwidth utilization of each subroutine and each loop nest. Hardware counters are used to estimate the total amount of memory transfer. The table in Figure 6.9 lists the effective memory bandwidth of seven major subroutines, which represents 95% of overall running time.

Subroutines	Achieved BW	BW Utilization
compute_rhs	252MB/s	84%
x_solve	266MB/s	89%
y_solve	197MB/s	66%
z_solve	262MB/s	87%
lhsx	321MB/s	107% [1]
lhsy	279MB/s	93%
lhsz	96MB/s	32%

Figure 6.9 Memory bandwidth utilization of *NAS/SP*

The last column of the table in Figure 6.9 shows that all subroutines utilized 84% or higher memory bandwidth except *y_solve* and *lhsz*. The low memory bandwidth utilization prompted the need for user tuning. Subroutine *lhsz* had the largest potential gain for performance tuning. The subroutine has three loop nests, all had normal bandwidth utilization except the first one, which had an extremely low bandwidth utilization of less than 11%. The inspection by a user (myself) revealed that the problem was due to excessive TLB misses. Manual application of array expansion and loop permutation was able to eliminate a large portion of the TLB misses and improve the running time of the loop nest by a factor of 5 and the overall execution time by over 15%.

---

[1] Pure data-copying loops with little computation can achieve a memory bandwidth that is slightly higher than 300MB/s on SGI Origin2000.

Similar tuning process was then performed on *compute_rhs*, which had an average bandwidth utilization of 84%. However, not all loops performed well in *compute_rhs*. The examination of loop-level bandwidth utilization found two loops that utilized 65% and 44% of memory bandwidth because of cache conflicts in L1. Loop distribution and array padding were applied by hand. The modifications improved the two loops by 9% and 24% individually and overall running time by another 2.4%. After the tuning of both *lhsz* and *compute_rhs*, the performance of *SP* was improved from 45.1 MFlops/s to 55.5 MFlops/s, a speedup of 1.19.

Bandwidth-based tuning is more accurate in locating performance problems than other tuning techniques because it monitors the most critical resource—memory bandwidth. For example, flop rates are not as effective. The flop rates of the previously mentioned two loops in *compute_rhs* are over 30 MFlop/s before tuning, which are not much lower than other parts of the program. For example, all loops in *lhsx* have a flop rate fewer than 18 MFlop/s. By comparing the flop rates, a user may draw the wrong conclusion that the loops in *lhsx* are better candidates for tuning. However, the loops in *lhsx* cannot be improved because they already saturate the available memory bandwidth. Their flop rates are low because they are data-copying loops with little computation.

The successful tuning of *SP* shows that the automatic tuning support is extremely effective for a user to correct performance problems in large applications. Although there were over 80 loop nests in *SP*, bandwidth-based tuning automatically located three loop nests for performance tuning. As a result, we as programmers only needed to inspect these three loops, and simple source-level changes improved overall performance by 19%. In other words, the bandwidth-based tuning tool allowed us to obtain 19% of overall improvement by examining less than 5% of the code.

Performance Prediction

Bandwidth-based performance prediction approximates program-running time with the estimated memory-transfer time, that is, the total amount of memory transfer divided by the memory bandwidth of the machine. This section examines the accuracy of this prediction technique on the *SP* benchmark. Since the prediction requires estimation on the amount of memory transfer, the experiment will first measure it with hardware counters and then apply compiler analysis by hand to verify the accuracy of the compiler-based estimation.

The table in Figure 6.10 lists the actual running time of a single iteration of *SP*, the predicted time and the percent of error. The predicted time is the total amount of memory transfer divided by memory bandwidth. The prediction is given both with and without considering the effect of TLB misses in the first loop of *lhsz*, discussed in the previous section on user tuning. The table lists two predictions, the first assumes full memory bandwidth utilization for the whole program, and the other assumes an average utilization of 90%.

Computation	Exe Time	Pred. Time I Util=100%	Err. I	Pred. Time II Util=90%	Err. II
adi w/o TLB est.	59.0s	43.8s	-26%	48.6s	-18%
adi w TLB est.	59.0s	50.9s	- 14%	55.7s	- 5.6%
adi w/o *lhsz*	47.0s	40.0s	- 15%	44.3s	- 5.7%

Figure 6.10 Actual and predicted execution time

The first row of table in Figure 6.10 gives the estimation without considering the extra overhead of TLB misses in *lhsz*. The TLB overhead can be easily predicted by multiplying the number of TLB misses with full memory latency (338ns according to what is called restart latency in [HL97]), which adds to a total of 7.1 seconds. The second row gives the performance prediction considering this TLB overhead. The third row predicts performance for the program without *lhsz* (the rest represents over 80% of the overall execution time).

The third and fifth column of the table in Figure 6.10 show the error of prediction. When assuming full bandwidth utilization, the prediction error is 26% when not considering the abnormal TLB overhead, 14% when considering the TLB overhead, and 15% for the program without *lhsz*. When the assumed utilization is 90%, the prediction error is 18% when not considering TLB overhead, 5.6% when including the TLB cost, and 5.7% for the program without *lhsz*. The table shows that, with the estimation of the TLB cost and the assumption of 90% memory-bandwidth utilization, bandwidth-based prediction is very accurate, with an error of less that 6%. The similar errors in the last two rows also suggest that our static estimation of the TLB overhead is accurate.

The above predictions measured the amount of memory transfer through hardware counters. This is undesirable because we should predict program performance without running the program. So the next question is how accurate is the static estimation of a compiler. I manually applied a simplified version of the data analysis described in Section 5.2. In fact, only the bounded-section analysis was used, which counted only the number of capacity misses in each loop nest. I did not expect many conflict misses because the L2 cache on SGI Origin2000 is two-way set associative and 4MB in size.

Two subroutines were manually analyzed: *compute_rhs* and *lhsx*, which consisted of 40% of the total running time. Subroutine *compute_rhs* had the largest code source and the longest running time among all subroutines. It also resembled the mixed access patterns in the whole program because it iterated the cubic data structure through three directions. The subroutine *lhsx* accessed memory contiguously in a single stride. The following table lists the actual memory transfer measured by hardware counters, the predicted memory transfer by the hand analysis, and the error of the static estimation.

Subroutine	Actual	Predicted	Error
lhsx	396MB	406MB	+ 3%
compute_rhs	5308MB	5139MB	- 3%

Figure 6.11 Actual and predicted data transfer

The errors shown in the third column of the table in Figure 6.11 are within 3%, indicating that the static estimation is indeed very accurate. Assuming this accuracy holds for other parts of *SP*, the bandwidth-based analysis tool could predict the overall performance within an error of less than 10%, assuming an average memory bandwidth utilization of 90%.

6.6 Summary

Global optimizations The two-step global strategy of loop fusion and data regrouping are extremely effective for the benchmark programs tested, improving overall speed by 14% to a factor of 2.33 for kernels and a factor of 1.5 for the full application. Furthermore, the improvement is obtained solely through automatic source-to-source compiler optimization. The success especially underscores the following three important aspects:

- Aggressive loop fusion. All test programs have loops with a different number of dimensions. Mere loop alignment cannot fuse any of the tested programs except for a few loops in *SP*. *Swim* also requires loop splitting.

- Conservative data regrouping. Data regrouping improves cache and program performance in most cases. The only few degradations are due to the side effects of data regrouping on the back-end compiler. These problems can be easily corrected if the data transformation is made by the back-end compiler itself. Therefore, data regrouping should always be beneficial in practice.

- Combined optimization strategy of computation fusion and data regrouping. Although when used together they achieve substantial performance improvement, neither can do so when used alone. In fact, loop fusion degrades performance in most cases if used without data regrouping, and data regrouping sees little or no opportunity without loop fusion, especially for large programs.

Data regrouping is also very beneficial for dynamic applications. Although these programs have unpredictable data access within each array, the relations among multiple arrays are consistent and can be determined by a compiler. Consequently, data regrouping is able to improve global cache reuse despite unknown and dynamic data access patterns. For the two dynamic applications where data regrouping is applied, it reduces 17% to 70% of cache and TLB misses and improves performance by factors 1.3 and 1.9.

Dynamic optimizations Run-time data packing is very effective for dynamic programs whose access pattern remains unknown until run time and changes during the execution. By analyzing and optimizing data layout at run time, data packing reduces the number of L2 misses by 21% to 84% and the number of TLB misses by 33% to 97%, and as a result, improves overall program performance by a factor up to 4.36. The run-time cost of data reorganization is negligible, consuming only 0.14% to 0.7% of the overall execution time. The overhead due to data indirections through run-time data maps is significant, but it can be effectively eliminated by the two packing optimizations, pointer update and array alignment. The two optimizations improve performance by factors ranging from 1.08 to 5.56. In addition, the run-time optimizations and global data regrouping complement each other and achieve the best performance when both are used together.

Bandwidth-based performance tool Bandwidth-based tuning and prediction is simple yet very accurate. When evaluated on the 3000-line NAS *SP* benchmark, it enables a user to obtain an overall speedup of 1.19 by inspecting and tuning only 5% of the program code. The compile-time prediction on the whole-program execution time can be within 10% difference of the actual execution time.

Chapter 7

Conclusions

"To travel hopefully is a better thing than to arrive, and the true success is to labour." – Rober Louis Stevenson (1850-1894)

At the outset, this dissertation demonstrated the serious performance bottleneck caused by insufficient memory bandwidth. From then on, it has taken the goal of minimizing memory-CPU communication through compiler optimizations. This chapter first summarizes the new techniques that have been developed in the preceding chapters. Then it discusses future extensions of this work. Finally, the chapter concludes with the final remarks restating the underlying theme of this dissertation.

7.1 Compiler Optimizations for Cache Reuse

The main contribution of this dissertation is a set of new compiler transformations that optimizes cache performance both at the global level and at run time.

Global optimizations Global optimizations include computation fusion and data grouping. Chapter 2 describes two algorithms for computation fusion: *reuse-driven execution* measures the limit of global cache reuse on an ideal machine, and more importantly, *maximal loop fusion* realizes most of the global benefit on a real machine. Maximal loop fusion fuses data-sharing statements whenever possible, achieves bounded reuse distance within a fused loop, and maximizes the amount of fusion for multi-dimensional loops. While maximal loop fusion improves temporal cache reuse for the whole program, *inter-array data regrouping* maximizes spatial cache reuse for the entire data. Inter-array regrouping, presented in Chapter 3, merges data at a hierarchy of granularity from large array segments to individual array elements. It makes data access as contiguous as possible and it eliminates unnecessary memory writebacks. Both maximal loop fusion and inter-array data grouping are fast; their time complexity is approximately $O(V * A)$, where V is the length of the program and A is the number of data structures in the program.

Maximal loop fusion and inter-array data regrouping are currently the most powerful set of global transformations in the literature. Maximal loop fusion is more aggressive than previous loop fusion techniques because it can fuse all statements in a program whenever permitted by data dependence. Data regrouping is the first to split and regroup global data structures and to do so with guaranteed profitability and compile-time optimality. The overall strategy is also the first in the literature to combine a global computation transformation with a global data transformation.

Dynamic optimizations Dynamic optimizations include *Locality grouping* and *data packing*, presented in Chapter 4. They improve dynamic cache reuse by reordering irregular computation and data at run time. Locality grouping merges computations involving the same data, and data packing groups data used by the same computation. Both are general-purpose transformations. Locality grouping reorders any set of independent computations, data packing reorganizes any non-contiguous data access. In addition, both transformations incur a minimal run-time overhead, which is linear in time and space.

More importantly, locality grouping and data packing are the first set of dynamic optimizations that are automatically inserted and optimized by a compiler. The basis for this automation is *compiler indirection analysis*, which analyzes indirect data access in a program and identifies all opportunities of data reorganization. Two compiler optimizations, *pointer update* and *array alignment* are used to remove data indirections after data relocation. Both optimizations are extremely effective in removing the overhead of run-time data transformation.

Performance model and tool The *balance*-based performance model, described in Chapter 1, is the first to consider the balance of bandwidth resources at all levels of a computing system from the CPU flop rate to memory bandwidth. The balance-based model has clearly demonstrated the existence of the memory bandwidth bottleneck and its constraint on performance. Based on this model, Chapter 5 designed a *bandwidth-based tool*. The new tool supports effective user tuning by automatically locating performance problems in large applications. In addition, the tool provides accurate performance prediction.

Summary of evaluation results The experimental evaluation has verified that the new global strategy achieved dramatic reductions in the volume of data transferred for the programs studied. The table in Figure 7.1 compares the amount of data transferred for versions of each program with no optimization, with optimizations provided by the SGI compiler, and after transformation via the strategy developed in this dissertation. If we compare the average *reduction* in misses due to compiler techniques, the new strategy, labeled by column *New*, does significantly better than the SGI compiler, labeled by column *SGI*.

program	L1 misses			L2 misses			TLB misses		
	NoOpt	**SGI**	**New**	**NoOpt**	**SGI**	**New**	**NoOpt**	**SGI**	**New**
Swim	1.00	1.26	1.15	1.00	1.10	0.94	1.00	1.60	1.05
Tomcatv	1.00	1.02	0.97	1.00	0.49	0.39	1.00	0.010	0.010
ADI	1.00	0.66	0.40	1.00	0.94	0.53	1.00	0.011	0.005
NAS/SP	1.00	0.97	0.77	1.00	1.00	0.49	1.00	1.09	0.67
average	**1.00**	**0.98**	**0.82**	**1.00**	**0.88**	**0.59**	**1.00**	**0.68**	**0.43**
Moldyn	1.00	1.15	0.34	1.00	0.99	0.19	1.00	0.77	0.10
Mesh	1.00	1.02	0.47	1.00	1.34	0.39	1.00	0.57	0.57
Magi	1.00	1.15	0.82	1.00	1.25	0.76	1.00	1.00	0.36
NAS/CG	1.00	1.01	0.58	1.00	0.95	0.15	1.00	0.97	0.03
average	**1.00**	**1.08**	**0.55**	**1.00**	**1.13**	**0.37**	**1.00**	**0.83**	**0.27**

Figure 7.1 Summary of evaluation results

On average for the four regular applications measured, the new strategy outperforms the SGI compiler by factors of 9 for L1 misses, 3.4 for L2 misses, and 1.8 for TLB misses. The improvement is even larger for the four irregular and dynamic programs, where both global and dynamic optimizations are applied. On average, the new strategy reduces L1 misses by 45%, L2 misses by 63%, and TLB misses by 73%. In contrast, the SGI compiler causes an average of 8% more L1 misses and 13% more L2 misses for these dynamic applications. Thus, the global and dynamic strategy developed in this dissertation has a clear advantage over the more local and static strategies employed by an excellent commercial compiler.

7.2 Future Work

The successful development of global and dynamic optimizations has opened new fascinating opportunities for future compiler research. New improvement can come from overcoming the overhead of global and dynamic optimizations and from extending their capabilities. This section touches on research directions that are important and can be approached by direct extensions of this work.

Storage optimization after fusion Extensive loop fusion enables new opportunities for storage optimization. Section 2.5 of Chapter 2 has given two examples of storage reduction and store elimination. These two techniques need to be developed and evaluated on real programs. In general, computation fusion provides more freedom in organizing data and its uses. The data transformations mentioned here are just the tip of an iceberg.

Improving the fusion heuristic Although the benefit of maximal fusion has been verified, it remains unknown how much more benefit can be obtained by improving over the greedy heuristic currently used. How can data sharing be reduced among fused loops? How well do other heuristics perform, especially the one of always fusing along the heaviest edge by Kennedy[Ken99]? Further evaluation is required to find the best fusion method.

Register allocation after fusion Since loop fusion may merge too much computation in a fused loop, it may overflow machine registers and dramatically increase the number of register loads and stores. A direct remedy exists, which is to distribute a large loop into smaller ones that do not overflow registers. The distribution is in fact a form of fusion after computations are divided into the smallest unit. The fusion is equivalent to a fixed-size partitioning of hyper-graphs and is NP-hard. The problem is similar to loop fusion because it needs to minimize data sharing; however, it is also different because it fuses loops up to a fixed size.

Data grouping for arbitrary programs In this dissertation, array regrouping uses compiler analysis to identify computation phases and data access patterns. When compiler analysis is not available, data regrouping can still use profiling analysis. By profiling the execution, it can define computation phases as time intervals in which the amount of data access is larger than cache. If two data items, as for example, two members of different object classes, are always accessed together, they can then be grouped into the same cache block. In this way, data grouping can be applied to arbitrary programs.

Data grouping for parallel programs On shared memory machines, cache blocks are the basis of data consistency and consequently the unit of communication among parallel processors. Data regrouping can be applied to parallel programs to improve cache-block utilization, which leads to reduced communication latency and increased communication bandwidth.

Indirection analysis for object-oriented programs Indirection analysis can be extended to dynamic allocated objects linked by pointers. It can be done by analyzing the relation of objects based on their types and then recording the access sequence of related objects at run time. Such an extension would allow data packing to be applied to object-oriented programs.

Data reuse analysis for parallelization The reuse-distance analysis is a general tool that can study unconventional optimizations by examining their effect on data reuse. One important direction is to experiment with innovative methods of program parallelization and data communication.

Automatic run-time optimizations The efficient performance monitoring technique, developed in Chapter 5, has made it possible that the execution status of a program can be monitored and its performance problems can be identified and corrected dynamically. This adaptability is extremely important for large programs running on heterogeneous machines where a single program code cannot work well everywhere.

7.3 Final Remarks

The dissertation can be viewed as a pursuit of two goals. The first, a fundamental one, is to minimize the memory-CPU communication by caching. The second, a practical concern, is to avoid losing software productivity in the search for higher performance. This research has found a middle ground to balance between these two goals. It optimizes the whole program at all times but it does so with automatic methods that are transparent to a programmer. This dual theme of optimization and automation permeates this dissertation and extends to its vision of the future. As the world is becoming a ubiquitously connected computing environment, programming tasks in the future will be far more complex and difficult because of the scale of the software and the complexity of hardware. Today's manual process of programming is unlikely to meet this future challenge. The extension of this work may offer a better alternative through programming automation. In fact by providing powerful techniques for global and dynamic program transformation, this research has paved the way for the automation of complex and large-scale programming tasks, thus making software development less labor-intensive and more manageable.

"He who knows others is learned. He who knows himself is wise." – Lao Tzu (about 500 BC)

Bibliography

[AC72] F. Allen and J. Cocke.
A catalogue of optimizing transformations.
In J. Rustin, editor, *Design and Optimization of Compilers*. Prentice-Hall, 1972.

[AFR98] I. Al-Furaih and S. Ranka.
Memory hierarchy management for iterative graph structures.
In *Proceedings of IPPS*, 1998.

[AK87] J. R. Allen and K. Kennedy.
Automatic translation of Fortran programs to vector form.
ACM Transactions on Programming Languages and Systems, 9(4):491–542, October 1987.

[All83] J. R. Allen.
Dependence Analysis for Subscripted Variables and Its Application to Program Transformations.
PhD thesis, Dept. of Computer Science, Rice University, April 1983.

[ASKL81] W. Abu-Sufah, D. Kuck, and D. Lawrie.
On the performance enhancement of paging systems through program analysis and transformations.
IEEE Transactions on Computers, C-30(5):341–356, May 1981.

[Bai92] D. Bailey.
Unfavorable strides in cache memory systems.
Technical Report RNR-92-015, NASA Ames Research Center, 1992.

[BFKK91] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer.
A static performance estimator to guide data partitioning decisions.
In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[BGK96] D. C. Burger, J. R. Goodman, and A. Kagi.
Memory bandwidth limitations of future microprocessors.
In *Proceedings of the 23th International Symposium on Computer Architecture*, 1996.

[Cal87] D. Callahan.
A Global Approach to Detection of Parallelism.

PhD thesis, Dept. of Computer Science, Rice University, March 1987.

[Car92] S. Carr.
Memory-Hierarchy Management.
PhD thesis, Dept. of Computer Science, Rice University, September 1992.

[CCC$^+$97] R. Chandra, D. Chen, R. Cox, D.E. Maydan, and N. Nedeljkovic.
Data distribution support on distributed shared memory multiprocessors.
In *Proceedings of '97 Conference on Programming Language Design and Implementation*, 1997.

[CCJA98] B. Calder, K. Chandra, 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)*, San Jose, Oct 1998.

[CCK88] D. Callahan, J. Cocke, and K. Kennedy.
Estimating interlock and improving balance for pipelined machines.
Journal of Parallel and Distributed Computing, 5(4):334–358, August 1988.

[CDL99] T.M. Chilimbi, B. Davidson, and J.R. Larus.
Cache-conscious structure definition.
In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, 1999.

[CK89] S. Carr and K. Kennedy.
Blocking linear algebra codes for memory hierarchies.
In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, IL, December 1989.

[CKP90] D. Callahan, K. Kennedy, and A. Porterfield.
Analyzing and visualizing performance of memory hierarchies.
In *Performance Instrumentation and Visualization*, pages 1–26. ACM Press, 1990.

[CL95] M. Cierniak and W. Li.
Unifying data and control transformations for distributed share d-memory machines.
In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, 1995.

[CQ93] Mark J. Clement and Michael J. Quinn.
Analytical Performance Prediction on Multicomputers.
In *Proceedings of Supercomputing'93*, November 1993.

[Dar99] Alain Darte.
On the complexity of loop fusion.
In *Proceedings of International Conference on Parallel Architecture and Compilation*, pages 149–157, Newport Beach, CA, Oct 1999.

[DJP+92] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis.
The complexity of multiway cuts.
In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, May 1992.

[DK00] C. Ding and K. Kennedy.
Memory bandwidth bottleneck and its amelioration by a compiler.
In *Proceedings of the 2000 International Parallel and Distributed Processing Symposium*, Cancun, Mexico, May 2000.

[DMS+92] R. Das, D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy.
The design and implementation of a parallel unstructured euler solver using software primitives.
In *Proceedings of the 30th Aerospace Science Meeting*, Reno, Navada, January 1992.

[DUSH94] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang.
Communication optimizations for irregular scientific computations on distributed memory architectures.
Journal of Parallel and Distributed Computing, 22(3):462–479, September 1994.

[FST91] J. Ferrante, V. Sarkar, and W. Thrash.
On estimating and enhancing cache effectiveness.
In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

[GH93] Aaron J. Goldberg and John L Hennessy.
Mtool: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications.
IEEE Transactions on Parallel and Distributed Systems, 4(1), 1993.

[GJG88] D. Gannon, W. Jalby, and K. Gallivan.
Strategies for cache and local memory management by global program transformation.
Journal of Parallel and Distributed Computing, 5(5):587–616, October 1988.

[GOST92] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath.

Collective loop fusion for array contraction.
In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[HK91] P. Havlak and K. Kennedy.
An implementation of interprocedural bounded regular section analysis.
IEEE Transactions on Parallel and Distributed Systems, 2(3):350–360, July 1991.

[HL97] Cristina Hristea and Daniel Lenoski.
Measuring memory hierarchy performance of cache-coherent multiprocessors using micro benchmarks.
In *Proceedings of SC97: High Performance Networking and Computing*, 1997.

[hSM97] harad Singhai and Kathryn S. McKinley.
A parameterized loop fusion algorithm for improving parallelism and cache locality.
The Computer Journal, 40(6):340–355, 1997.

[HT98] H. Han and C.-W. Tseng.
Improving compiler and run-time support for adaptive irregular codes.
In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 1998.

[JE95] Tor E. Jeremiassen and Susan J. Eggers.
Reducing false sharing on shared memory multiprocessors through compile time data transformations.
In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188, Santa Barbara, CA, July 1995.

[KAP97] Induprakas Kodukula, Nawaas Ahmed, and Keshav Pingali.
Data-centric multi-level blocking.
In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, June 1997.

[KCRB98] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee.
A matrix-based approach to the global locality optimization problem.
In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 1998.

[Ken99] K. Kennedy.
Fast greedy weighted fusion.
Technical Report CRPC-TR-99789, Center for Research on Parallel Computation (CRPC), 1999.

[KH78] D. G. Kirkpatrick and P. Hell.

On the completeness of a generalized matching problem.
In *The Tenth Annual ACM Symposium on Theory of Computing*, 1978.

[KM93] K. Kennedy and K. S. McKinley.
Typed fusion with applications to parallel and sequential code generation.
Technical Report TR93-208, Dept. of Computer Science, Rice University, August 1993.
(also available as CRPC-TR94370).

[Kre95] U. Kremer.
Automatic Data Layout for Distributed Memory Machines.
PhD thesis, Dept. of Computer Science, Rice University, October 1995.

[LRW91] M. Lam, E. Rothberg, and M. E. Wolf.
The cache performance and optimizations of blocked algorithms.
In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Santa Clara, CA, April 1991.

[MA97] N. Manjikian and T. Abdelrahman.
Fusion of loops for parallelism and locality.
IEEE Transactions on Parallel and Distributed Systems, 8, 1997.

[McC95] John D. McCalpin.
Sustainable memory bandwidth in current high performance computers.
http://reality.sgi.com/mccalpin_asd/papers/bandwidth.ps, 1995.

[MCF99] Nicholas Mitchell, Larry Carter, and Jeanne Ferrante.
Localizing non-affine array references.
In *Proceedings of International Conference on Parallel Architecture and Compilation*, Newport Beach, CA, Oct 1999.

[McI97] Nathaniel McIntosh.
Compiler Support for Software Prefetching.
PhD thesis, Rice University, Houston, TX, July 1997.

[MCT96] K. S. McKinley, S. Carr, and C.-W. Tseng.
Improving data locality with loop transformations.
ACM Transactions on Programming Languages and Systems, 18(4):424–453, July 1996.

[MCWK99] John Mellor-Crummey, David Whalley, and Ken Kennedy.
Improving memory hierarchy performance for irregular applications.
In *Proceedings of the 13th ACM International Conference on Supercomputing*, pages 425–433, Rhodes, Greece, 1999.

[ML98] Philips J. Mucci and Kevin London.
The cachebench report.

Technical Report ut-cs-98-394, University of Tennessee, 1998.

[Mow94] T. Mowry.
Tolerating Latency Through Software Controlled Data Prefetching.
PhD thesis, Dept. of Computer Science, Stanford University, March 1994.

[MT96] Kathryn S. McKinley and Olivier Temam.
A quantitative analysis of loop nest locality.
In *Proceedings of Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Boston, MA, Oct 1996.

[Por89] A. Porterfield.
Software Methods for Improvement of Cache Performance.
PhD thesis, Dept. of Computer Science, Rice University, May 1989.

[PR99] W. Pugh and E. Rosser.
Iteration space slicing for locality.
In *Proceedings of the Twelfth Workshop on Languages and Compilers for Parallel Computing*, August 1999.

[Pug92] W. Pugh.
A practical algorithm for exact array dependence analysis.
Communications of the ACM, 35(8):102–114, August 1992.

[SL99] Y. Song and Z. Li.
New tiling techniques to improve cache temporal locality.
In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, 1999.

[SZ98] M. L. Seidl and B. G. Zorn.
Segregating heap objects by reference behavior and lifetime.
In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, Oct 1998.

[TA94] K. A. Tomko and S. G. Abraham.
Data and program restructuring of irregular applications for cache-coherent multiprocessors.
In *Proceedings of '94 International Conference on Supercomputing*, 1994.

[Tha81] K. O. Thabit.
Cache Management by the Compiler.
PhD thesis, Dept. of Computer Science, Rice University, 1981.

[WL91] M. E. Wolf and M. Lam.
A data locality optimizing algorithm.
In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[Wol82] M. J. Wolfe.
Optimizing Supercompilers for Supercomputers.
PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1982.