Optimizing Programs by Data and Control Transformations

by

Michał Cierniak

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Professor Wei Li

Department of Computer Science
The College
Arts and Sciences

University of Rochester
Rochester, New York

1997

Curriculum Vitae

Michał Cierniak was born in Katowice, Poland on January 12, 1967. He attended the Technical University of Silesia in Gliwice, Poland from 1985 to 1990 and graduated with the degree of Magister in 1990. From 1990 to 1991 he attended the University of Edinburgh in Edinburgh, Scotland with the support of the TEMPUS scholarship and graduated with a Master of Science degree in 1991. He came to the University of Rochester in the Fall of 1992 and began graduate studies in computer science. He pursued his research in *Data and Control Optimizations for Cache Locality* under the direction of Professor Wei Li and received the Master of Science degree in 1994.

Acknowledgments

I would like to thank my advisor, Wei Li, for guiding me through my research. Without his help, especially in my early years at Rochester, I would not have been able to decide which directions are worth pursuing and what is important in academic research.

I also want to thank other members of my committee: Alexander Albicki, Tom LeBlanc and Michael Scott whose suggestions have improved this dissertation.

During my stay at Rochester I have also received valuable help in my research from: Ricardo Bianchini, Sandhya Dwarkadas, Galen Hunt, Leonidas Kontothanassis, Wagner Meira, Srinivasan Parthasarathy, Suresh Srinivas, Robert Stets, Jack Veenstra, and Mohammed Zaki.

My friends at the Computer Science Department: students, staff and faculty created a great environment for work and for fun.

My friends not directly affiliated with the Computer Science Department: Melissa, Greg, Jeanne, Jules, Joanna, Maja, Jim, Kasia and many others—have made my life after work enjoyable. Thanks! I wish I had more time to spend with all of you.

The support of my family and especially my mother, Czesława, made it all possible.

I would like to thank the developers of Polaris [10] and Kaffe [68] for providing excellent tools for compiler experiments.

This work was supported in part by an NSF Research Initiation Award (CCR-9409120) and ARPA contract F19628-94-C-0057.

Abstract

This dissertation presents new techniques designed to speed up the execution of computer programs by improving their memory locality. Locality is an important property for today's machines, because it hides the relatively high latency of computer memories.

Our techniques change the layout of multidimensional arrays by applying *data transformations*. We unify data transformations with *code transformations* which change the order of execution of loop nests. We solve related problems which would have been obstacles to the practical use of our techniques: we show how to detect and reduce array *overlapping* and how to recover structure from linearized arrays. Our optimizations reduce the execution times of sequential, scientific benchmarks by up to 50% over what is possible with previous techniques. Parallel programs are improved by as much as a factor of four.

In addition to implementing our techinques in a standard, *off-line*, compiler, we adapt our optimizations to Just-In-Time (JIT) compilation. The JIT translation becomes very important with the increasing popularity of mobile technologies such as Java. We argue that new, faster algorithms are needed in that context. We propose a collection of fast, approximate compiler techniques for data transformations and show that they are effective for Java programs.

Table of Contents

Curriculum Vitae **ii**

Acknowledgments **iii**

Abstract **iv**

List of Figures **vii**

1 Introduction **1**

1.1 The Problem 1

1.2 The Solution 2

1.3 Contributions 3

1.4 Outline 4

2 Array Transformations **5**

2.1 Related Work 5

2.2 Motivation 6

2.3 Algebraic Representation of Data Mappings 8

2.4 Locality Model 14

2.5 Data Transformations 18

2.6 Experiments 20

3 Unifying Array and Loop Transformations **24**

3.1 Motivation 25

3.2 The Unified Approach 27

3.3 Experiments 30

4 Data Structure Recovery **36**
4.1 Motivation 37
4.2 Framework 39
4.3 Rectangular Arrays 41
4.4 Non-rectangular Arrays 46
4.5 Multiple References 51

5 Interprocedural Array Transformations **55**
5.1 Motivation 55
5.2 Array Overlap Graph 57
5.3 Improving Accuracy by Procedure Cloning 59
5.4 Coercing the Types of Overlapping Arrays 63
5.5 Code Structure Recovery 67
5.6 Experimental Results 68

6 Just-In-Time Optimizations for Java **70**
6.1 Motivation 71
6.2 Overview of Array Transformations 72
6.3 Off-line Optimizations 74
6.4 Just-In-Time Optmizations 83
6.5 Experiments 90

7 Conclusions **94**
7.1 Summary 95
7.2 Future Work 97

Bibliography **99**

List of Figures

1.1 Row-major layout 2
1.2 Column-major layout 2

2.1 Loop interchange example 7
2.2 Row-major layout 16
2.3 Column-major layout 16
2.4 Array reference example 16
2.5 Transitive closure: effect of data transformations 21
2.6 Improvements in execution times for selected applications. 22
2.7 Transitive closure: classification of memory accesses 23

3.1 MxMxM (multiplication of three matrices): original code 26
3.2 MxMxM: control and data transformations 27
3.3 Array reference example after loop interchange 28
3.4 A synthetic benchmark 30
3.5 Execution times for the synthetic benchmark 31
3.6 Speedups for the synthetic benchmark 32
3.7 Execution times for the MxMxM program 33
3.8 Execution times for the economics program 33
3.9 Normalized miss numbers for MxMxM 34
3.10 Reference distances for the economics program. 35
3.11 Reference distances for MxMxM. 35

4.1 A loop nest from TRFD/OLDA 38
4.2 A 2-D array reference 39
4.3 A linearized reference 41

4.4 Non-rectangular loop 46
4.5 Linearized Loop Structure 52
4.6 An example from apsi 53

5.1 Aliasing vs. overlapping. 57
5.2 Example which shows that the analysis may be inaccurate. 59
5.3 Array Overlap Graph 59
5.4 Code fragment from `CHOLSKY`. 60
5.5 Array Overlap Graph before cloning 60
5.6 Code fragment from `CHOLSKY` after cloning. 61
5.7 Array Overlap Graph after cloning 61
5.8 Code fragment from apsi before type coercion. 65
5.9 Code fragment from apsi after type coercion. 65
5.10 Procedure `COPY1` *without* code structure recovery. 67
5.11 Procedure `COPY1` *with* code structure recovery. 67
5.12 Speedups for selected benchmarks 69

6.1 Internal representation of a Java array `double[n][m]` 73
6.2 Class hierarchy of the nodes in the JavaIR intermediate representation (only selected expression and statement types are shown) . 75
6.3 A store during expression evaluation 80
6.4 A control flow graph for a short-circuit operator 80
6.5 Code generated for `a[i][j]` by Kaffe for the i86 architecture . . . 91
6.6 Speedups achieved on a Pentium Pro/200 PC 92
6.7 Break-down of the optimization time for cholesky 93
6.8 Briki vs. kaffe 93

1 Introduction

1.1 The Problem

Many computer applications are limited by the speed available with current technology. The slow speed of random access memories relative to processor speed is one of major bottlenecks in today's computers. Program execution can be sped up by the use of cache memories (*caches*). Caches are smaller and faster than main memories and they work by exploiting the *locality* of memory references. The effect of locality is that some memory references can be satisfied from the cache rather than from main memory. It is widely accepted that caches improve the performance of many applications and virtually all modern microprocessors have either built-in caches or can work with an external cache.

Many applications do not fully utilize caches and the performance of those applications can be significantly improved by program restructuring. Programs can be restructured manually or automatically. Manual optimizations have several disadvantages:

- They have to be performed by programmers who have a very good understanding of both the machine architecture and program structure, and therefore manual optimizations are very difficult and time-consuming.
- Many cache-oriented transformations are machine dependent and would have to be repeated for every new computer that will run the application.
- Transformed programs are difficult to understand and maintain since the optimizations hide the real algorithm and data structures.

It is better to perform locality-enhancing optimizations automatically. Locality can be improved by the compiler or by the runtime system (usually the operating system). Both approaches have been employed in the past with different degrees of success. However, even the most sophisticated of the existing techniques do not fully improve the cache utilization of all applications and clearly new solutions to this problem are needed.

1.2 The Solution

The thesis of this dissertation is:

> Array restructuring can improve the performance of many real application over what is possible with other locality-improving techniques. Further improvement can be achieved by integrating array and control transformations.
>
> Unified array and control restructuring can be performed fully automatically by an optimizing compiler.

This section will explain, in high-level terms, what array restructuring (also referred to as *array transformations* or *data transformations*) is and how it can help increase the cache hit ratio. The rest of this dissertation will give more details, including formal definitions and algorithms needed to perform array transformations.

To understand what an array transformation is, consider the following code fragment.

```
real A[0..3, 0..2]
...
for j = 0, 2
  for i = 0, 3
    ...  A[i, j] ...
```

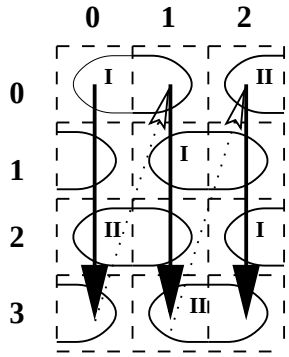

Figure 1.1: Row-major layout

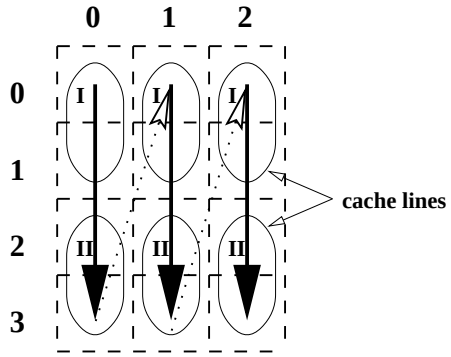

Figure 1.2: Column-major layout

Figures 1.1 and 1.2 present graphically two possible layouts for array `A` in memory for a hypothetical memory organization with each cache line holding two array elements. The cache is direct-mapped and can hold two cache lines. Memory areas marked with **I** map to the first line in the cache and memory areas marked

with **II** map to the second cache line. Arrows represent the order of memory accesses.

The two organizations of the two-dimensional array `A` result in very different locality for the reference `A[i,j]`. For our simple analysis, assume that there are no other memory references in our loop nest. By following the accesses to elements of `A` along the arrows in Figures 1.1 and 1.2 (assuming that initially none of the elements of `A` was in cache), we can determine that out of 12 memory references only two (accesses to `A[2,1]` and `A[1,2]`) are cache hits for the row-major layout. For the column-major layout exactly half of memory references are cache hits.

For this code fragment, the column-major layout of `A` results in a much higher cache hit ratio. If, for our hypothetical computer, the cost of cache hit was t_{hit} cycles and the cost of a miss was t_{miss}, the total time spent on accessing memory for the row-major layout would be $T_r = 10t_{\text{miss}} + 2t_{\text{hit}}$, time for column-major would be $T_c = 6t_{\text{miss}} + 6t_{\text{hit}}$. If we assumed the values of $t_{\text{hit}} = 2$ and $t_{\text{miss}} = 10$, the column-major layout would result in a 30% shorter total memory access time.

Our experiments show that the performance of many real programs can be improved significantly by choosing the right layout for their data structures. Moreover a large set of programs cannot be improved by any of other existing optimization techniques.

1.3 Contributions

The main contributions of this dissertation are:

- We propose *array transformations* as a new technique to improve cache locality of scientific applications. Array transformations can be used even when other techniques fail. For instance, explicitly parallel programs which cannot benefit from loop transformations, can be sped up with array transformations. Further, we identify issues which must be solved to change array layouts without affecting the semantics of popular programming languages. The most important problem preventing array transformations is *aliasing*—caused, for instance, by parameter passing. We define the property of *overlapping* by extending the traditional notion of aliasing to cover all aspects necessary for our optimizations. We not only show how to detect, efficiently represent and use the overlapping information, but we also design an algorithm which reduces overlapping by *selective procedure cloning*.

- We design an analytical framework and propose algorithms to *unify* array and loop transformations. We show that neither array nor loop transformations suffice to give the best possible performance for some scientific applications. Moreover, it is not sufficient to apply both loop and array

transformations separately in any order—a unified approach must be used for best results.

- We design very fast, approximate algorithms to perform array transformations in a Just-In-Time (JIT) compiler. JIT compilers are becoming ubiquitous in today's computing world due to the rising popularity of mobile code made possible with technologies like Java, Inferno or Omniware. We implement these algorithms in a Java JIT compiler and show that our algorithms are indeed fast and accurate enough for scientific benchmarks.

1.4 Outline

We introduce the concept of array transformations in Chapter 2. Chapter 3 shows how array transformations can be unified with loop transformations. Chapter 4 shows how a compiler can automatically recover the multidimensional structure of arrays which have been linearized by programmers. Techniques for ensuring legality of data transformations in the presence of procedure calls are shown in Chapter 5. Issues related to applying array transformations in a Just-In-Time compiler are discussed in Chapter 6. We conclude in Chapter 7.

2 Array Transformations

2.1 Related Work

Organization of multidimensional arrays was first studied for reducing access times of vector operations in multi-bank memories. For parallel computers with shared memory organized in separate memory modules, the effective memory bandwidth can be increased if no simultaneous accesses to the same module are being issued. Many parallel storage schemes have been proposed to address that problem. Budnik and Kuck [11] developed a skewed array storage scheme which uses a prime number of memory modules to reduce memory conflicts. Lawrie [41] proposed another solution employing the Omega Network. Liu *et al.* [50] developed an algorithm called Exchange-Expansion and showed how to combine different array storage schemes. Wijshoff [67] describes arbitrary skewing schemes for multidimensional arrays. He also discusses the compactness of skewing schemes. Solutions proposed in [11, 41, 50] work well for common access patterns: rows, columns, diagonals. Others have proposed storage schemes which can be used in programs exhibiting non-standard access patterns [29, 39, 44]. Proposed array organizations and algorithms for deciding parallel storage schemes are very different from array restructuring schemes needed for improving locality on contemporary machines.

While array transformations have not been used before to improve data locality, transformations of other data structures have been proposed—mostly to eliminate false sharing. The work by Eggers and Jeremiassen [27] and by Bianchini and LeBlanc [9] showed that for some programs, code and data restructuring can eliminate or reduce false sharing so that performance can be improved. However, these transformation techniques are all performed by hand on specific application programs. Dubois *et al.* [26] describe a hardware mechanism to eliminate misses due to false sharing, but at the expense of very large amounts of other communication.

Automatic data distribution for distributed message passing machines is also related to our approach. It has been investigated by Balasundaram *et al.* [6],

by Hudak and Abraham [35] for sequentially iterated parallel loops, by Knobe *et al.* [40] for SIMD machines, by Li and Chen [43] for *index domain alignment*, by Ramanujam and Sadayappan [58] who find communication-free partitioning of arrays in fully parallel loops, by Gupta and Banerjee [32] with a constraint-based approach, by Anderson and Lam [3] on data alignment and parallelism, by Chatterjee *et al.* [16] on array alignment and by Bau *et al.* [7] on a clean linear algebra solution to the alignment problem. However, the data mapping issues are sufficiently different on distributed shared memory machines. For example, neither data reuse nor false sharing is considered in any of the above approaches, since they are usually irrelevant for message passing distributed memory machines.

2.2 Motivation

We will illustrate the differences between control and data transformations before showing why data transformations (array transformations) are preferred in many cases. The program in Figure 2.1a will be used to show how control and data transformations can be used to improve performance. We have chosen the example so that either loop or array transformations can be used to improve its locality. However, for our work we are interested in applications which cannot benefit from loop transformations.

Assume that arrays `A` and `B` have the default column major layout. We can see that they both have poor locality in this loop nest.

2.2.1 Control Transformations

While no previous work proposes array transformations, loop transformations have been long proposed to solve the same problem. A good overview of locality enhancing loop transformations is given presented by Wolfe in [70]. The algorithms have been proposed by Wolf and Lam [69], Li [47], by Li and Pingali [46, 45], by Carr *et al.* [13], by Gannon *et al.* [30], and by Eisenbeis *et al.* [28].

Different transformations may be used to improve data locality. For the code fragment from Figure 2.1a, a simple loop interchange resulting in the code shown in Figure 2.1b, will improve the locality of data accesses.

2.2.2 Data Transformations

Data locality of the code fragment from Figure 2.1a, can also be improved by a data transformation. If, without changing the code, we decide to use row major layout for both arrays, we will again obtain the optimal locality as in Section 2.2.1.

```
real A[1000, 1000], B[1000,1000]
for i = 1, 1000
  for j = 1, 1000
    A[i, j] = 2 * B[i, j]
  endfor
endfor
```

(a) Original program

```
real A[1000, 1000], B[1000,1000]
for j = 1, 1000
  for i = 1, 1000
    A[i, j] = 2 * B[i, j]
  endfor
endfor
```

(b) After loop interchange

Figure 2.1: Loop interchange example

The way some of the current programs are written may be a serious obstacle in applying data transformations. Existing programming languages allow programs whose correctness relies on the particular array layout. Mechanisms that can be blamed for that include pointer arithmetic and creating aliases of different types by the use of common blocks or type casts.

Loop transformations have been traditionally used to improve the locality of scientific applications. However, loop transformations fail in many cases:

- Array transformations work for imperfectly nested loops. Although loop transformation can be applied to many instances of imperfect loop nests [59], they cannot be used for many complex loop nests. As long as the conditions of Section 2.2.2 are met, we can always change the mapping of an array.
- We can ignore data dependences. Again, if we make sure that none of the programming language constructs discussed in Section 2.2.2 (pointer arithmetic *etc.*) is used for the arrays we want to remap, data transformations are always legal.
- They can be used to optimize explicitly parallel programs. For programs with user level parallelism, loop transformations are generally not possible (except for the sequential portions of the program). The reason is that the programmer has already decided the parallelization. Often, synchronization primitives are inserted which seriously limit or make impossible any loop transformations.
- In languages with well defined exception mechanisms (like Java or C++), any reordering of loop iterations must take into account exception handlers which may depend on partial results being computed in specific order.

2.3 Algebraic Representation of Data Mappings

We will first present a formal representation of array mappings. The representation will be used in the locality model described in Section 2.4.

2.3.1 Representing data mappings

We consider data mappings for arrays of any dimensionality. We first discuss linear mappings and extend them later to affine mappings.

A *mapping* is a function from a vector of array subscripts to an offset from the start of the memory block allocated for the array. Let us define a *subscript vector* to be the vector of array subscripts, where the ith element in the vector

is the subscript from the ith dimension of the array reference. A linear mapping function is represented as a vector, where the product of the subscript vector and the mapping vector is the offset.

For example, consider again the array example introduced on page 2 and assume row-major mapping as illustrated in Figure 1.1. The compiler must generate code which given a reference to an array element, will calculate the offset of that element. For the reference `A[i, j]`, the offset is

$$\zeta = 3i + j.$$

We can compactly represent such a mapping function with a vector. The mapping vector for array `A` would be $m = \begin{pmatrix} 3 \\ 1 \end{pmatrix}$ and we would apply it to a reference represented with a subscript vector $S = \begin{pmatrix} i \\ j \end{pmatrix}$. The offset is simply the inner product of those two vectors:

$$\zeta = (i\ j) \begin{pmatrix} 3 \\ 1 \end{pmatrix} = 3i + j.$$

As another example, consider an array declared as

```
real A[0..n-1, 0..n-1];
```

For a reference to an array element `A[i, j]`, the subscript vector is $S = \begin{pmatrix} i \\ j \end{pmatrix}$.

A mapping corresponding to a row major organization is a vector $m = \begin{pmatrix} n \\ 1 \end{pmatrix}$

To apply the mapping, we compute the inner-product of the subscript vector and the mapping vector. For our example,

$$\zeta = S^T m = (i\ j) \begin{pmatrix} n \\ 1 \end{pmatrix} = in + j$$

which indeed corresponds to the row major mapping. Column major mapping would be represented as a vector $\begin{pmatrix} 1 \\ n \end{pmatrix}$, and would produce the offset $i + nj$.

Linear mappings as described above are convenient for locality analysis. The actual mappings used in code generation should be extended to include a constant offset in order to reference the correct location of the array element in memory.

Note that we can use the linear part of an affine mapping for locality analysis and use full affine mappings in the generated code. This is possible, because the

constant offset does not change the locality properties of array accesses if we do not consider inter-array interferences.

The obviuos way of considering the constant offset, $\mathcal{D}$, would be to simply add the appropriate value to the expression for the array element offset. For our examples so far we did not need to include the constant offset, because the subscripts started from zero, which implies $\mathcal{D} = 0$.

The constant offset can be easily computed from the array type. Assume that a d-dimensional array is declared as:
`real A`$[l_1 : u_1, l_2 : u_2, \ldots\ l_d : u_d]$

The constant offset can be computed as

$$\mathcal{D} = -\sum_{i=1}^{d} l_i m_i.$$

To illustrate this point, consider the Fortran convention with column-major mapping array subscripts that start from one rather than zero. The array `A` will now be declared:

```
real A[1..n, 1..n];
```

And the constant offset can be calculated as:

$$\mathcal{D} = -\sum_{i=1}^{d} l_i m_i = -(1+n).$$

The complete offset will now be:

$$\zeta = S^T m + \mathcal{D} = (i\ j) \begin{pmatrix} n \\ 1 \end{pmatrix} - (1+n) = in + j - n - 1.$$

For compact representation, we can include the constant offset in the mapping vector. Now, subscripts and mapping vectors for n-dimensional arrays will be comprised of $n+1$ elements. For the example above and the constant offset of 0, we have

$$(i\ j\ 1) \begin{pmatrix} n \\ 1 \\ 0 \end{pmatrix} = in + j.$$

This mapping vector describes arrays whose subscripts range from 0 to $n-1$, as in the programming language C. In Fortran arrays have column major organization and subscripts range from 1 to n, so the analogous expression is

$$(i\ j\ 1) \begin{pmatrix} 1 \\ n \\ -(n+1) \end{pmatrix} = n(j-1) + i - 1.$$

In general, for a d-dimensional array with the mapping vector of

$$m = \begin{pmatrix} m_1 \\ m_2 \\ \dots \\ m_d \end{pmatrix},$$

we define the *extended mapping vector* to have an extra element equal to the constant offset:

$$m = \begin{pmatrix} m_1 \\ \dots \\ m_d \\ \mathcal{D} \end{pmatrix}.$$

In addition to the mapping and subscript vectors we define the following terms:

- **Number of data dimensions** d is the number of dimensions of an array.
- An **offset** ζ is a number which for a given array reference describes the distance of the referenced array element from the start of the array. It can be computed as: $\zeta = S^T m$. If the subscripts do not start from zero, we may have to use extended mapping vectors to compute correct offsets.
- A **subscript range vector** w describes the ranges of subscripts expressions in an array. The subscript range vector can be computed from the lower and upper bounds for subscript expressions $w_i = (u_i - l_i + 1)$.
- An **access matrix** [45] A describes the subscripts of an array reference in terms of loop variables. Access matrices are described in Section 2.4.2.
- An **offset vector** δ contains constant parts of each subscript expression.

2.3.2 Constraints on mapping vectors

Unambiguity

A data mapping must satisfy certain conditions. The most important condition is that it is a one to one mapping. This is needed to ensure that every subscript vector corresponding to a legal array reference maps to a unique offset.

Not every mapping vector satisfies the unambiguity condition.

Theorem 2.1 *If a subscript range for a given dimension is greater than one, the corresponding element in the mapping vector must not be zero.*

$$w_i > 1 \Longrightarrow m_i \neq 0$$

Proof: Obvious.

□

As we show below, other constraints exist. Without loss of generality, we use two-dimensional arrays to illustrate the ideas. It is straightforward to generalize the results to arrays of any dimensionality.

Square matrices Let us consider the following declaration (in the rest of this section, we assume that subscripts' lower bound is 0)

```
real A[n, n];
```

Let the mapping vector be $m = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$.

Lemma 2.1 *If* $w = \begin{pmatrix} n \\ n \end{pmatrix}$ *then the mapping vector* $m = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$ *is such that at least one of* $|a|, |b|$ *must be greater than or equal to* n.

Proof: Let $a, b > 0$. We will show that at least one of a, b must be greater than or equal to n. Assume that $a < n$ and $b < n$. Then, the subscript vectors $S_1 = (b\ 0\ 1)^T$ and $S_2 = (0\ a\ 1)^T$ both map to the offset $ab + c$. We can see that assuming $a > 0$ and $b > 0$ and $S_1 = (b\ 0\ 1)^T$ and $S_2 = (0\ a\ 1)^T$ led to a contradiction. Vectors S_1 and S_2 are legal because $a, b < n$. Therefore at least one of a, b must be greater than or equal to n.

The condition on the sign of a and b can be relaxed. Rather than having $a, b > 0$, we only require $a, b \neq 0$. With this assumption, similar reasoning shows that at least one of $|a|, |b|$ must be greater than or equal to n.

□

Rectangular matrices Now consider the following array

```
real A[n, k];
```

Lemma 2.2 *If* $w = \begin{pmatrix} n \\ k \end{pmatrix}$ *then the mapping vector* $m = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$ *is such that either* $a \geq k$ *or* $b \geq n$.

Proof: Let the mapping vector be $m = (a\ b\ c)^T$. Let $a, b > 0$. We will show that we must have either $a \geq k$ or $b \geq n$. Assume that $a < k$ and $b < n$. Then, the subscript vectors $S_1 = (b\ 0\ 1)^T$ and $S_2 = (0\ a\ 1)^T$ both map to the offset $ab + c$. Vectors S_1 and S_2 correspond to legal references to the array `A` because a and b are both within bounds for the subscripts of `A`. Therefore we must have either $a \geq k$ or $b \geq n$.

□

This result can be generalized to matrices of any dimensionality.

Dense mappings

In most applications, we want the mappings to be dense. That is if the array contains n elements, we want offsets for all valid array references be in the range $0, \ldots, n-1$.

Note that the dense mapping is not required for correctness. On the other hand the unambiguity condition considered earlier must be satisfied for the mapping to be valid.

All references allowed Consider the following array of size $s = kn$ as an example

```
real A[n, k];
```

Theorem 2.2 *For a given constant offset c, there are exactly two legal, linear mappings that are dense. One of them, $(1\ n\ 0)^T$, corresponds to column major orientation, the other one $(k\ 1\ 0)^T$, corresponds to row major orientation.*

Proof: Let the mapping vector be $m = (a\ b\ c)^T$. Let us first assume for simplicity that $a, b > 0$ and $c = 0$. By Lemma 2.2 $a \geq k$ or $b \geq n$. Consider the reference `A[n-1][k-1]` for both cases. The offset ζ is $a(n-1) + b(k-1)$. The maximum offset is $\zeta_{\max} = nm - 1$.

- $a \geq k$: If $b \geq 2$, we have $\zeta \geq k(n-1) + 2k - 2 = k(n+1) - 2$. We see that $\zeta > \zeta_{\max}$ provided that $k > 2$. Therefore we must have $b < 2$, *i.e.*, $b = 1$ and $a = k$.
- $b \geq n$: Similarly, we must have $a = 1$ and $b = n$.

□

Similarly, it can be shown that for an d-dimensional array, there are exactly $d!$ legal, dense, linear mappings.

2.3.3 Banded matrices

If we have more information about the range of array subscripts we may be able to allocate less memory than indicated by the product of the sizes in all dimensions of the array.

One case that frequently appears in scientific applications is that of a banded matrix. A banded matrix has non-zero elements only along the diagonals of the matrix. A matrix A has lower bandwidth p if $A_{ij} = 0$ for $i > j + p$ and upper bandwidth q if $A_{ij} = 0$ for $j > i + q$. There is no need to store the zeros in memory. For a matrix like that we can allocated memory using one of the mappings described below.

Assume that we have the following declaration:

```
real A[n, n];
```

We can establish that this array contains a banded matrix by program analysis or by use of programmer annotations. Assume that array `A` has a lower bandwidth p and an upper bandwidth q.

If there are no accesses to elements outside the band defined by p and q, the following four mappings can be used:

- $(p + q,\ 1,\ p)^T$,
- $(1,\ p + q,\ q)^T$,
- $(1 - n,\ n,\ pn)^T$, and
- $(n,\ 1 - n,\ qn)^T$.

All of these mapping will result in memory being allocated for only the $n(p+q+1)$ non-zero elements.

2.4 Locality Model

We want to optimize programs for execution time. To exploit the memory hierarchy, data reuse has to be maximized. To simplify our discussion, we will consider computers with large main memory and smaller, but faster cache memory. *Cache hit ratio* [34] is one metric for quantifying data reuse. The hit ratio for a given run of a program is a non-trivial function of machine parameters, operating system policies, load on the machine and the access pattern of the program itself.

A more machine-independent metric, which can be used in compilers, is *reference distance*. We measure how many *different* cache lines are accessed between

two references to the same cache line. We observe that this *reference distance* can be used to guide program optimizations.

As a locality model, we propose a *stride vector* to measure the locality of array references in a loop nest. In Section 2.5, this model is used for data optimizations. In Section 3.2, we show how to integrate the stride vectors formalism with transformation matrices, which are commonly used to represent loop transformations.

2.4.1 Reference distance

We use reference distance as a metric of the quality of data locality. This metric is not as accurate for a given machine as the cache hit ratio, but it has the advantage of being relatively independent from machine parameters—it only depends on the cache line size.

Reference distance for a given memory access is defined to be the number of distinct cache lines accessed since the last access to the same cache line (or 0 if the cache line has not been accessed before).

To analyze the program behavior, we create a histogram for all interesting references. The goal of the locality optimization is to decrease the distances for critical references. Note that to decrease the distance for some references, we may have to increase the distance for others.

2.4.2 Stride vectors

Our approach to representing data reuse for different data mappings uses a new concept of a *stride vector* instead of the more traditional reuse vectors [69, 47]. Elements of the stride vector give us information about data reuse. If an element is 0, then this loop carries *temporal reuse*, if an element is less than the size of a cache line, then the loop has *spatial reuse*.

Figures 2.2 and 2.3 illustrate the concept of a stride vector. Consider again the very simple example of a 4×3 array shown on page 2 in Section 1.2. The application's performance is influenced most by the locality of the array references in the innermost loop. For iteration `j` of the outer loop, four array elements are accessed in the inner loop: `A[0,j]`, `A[1,j]`, `A[2,j]` and `A[3,j]`. Locality of those accesses can by summarized by a distance, or *stride*, between those elements. Figures 2.2 shows that for the row-major layout of `A`, the stride is 3, for the column-major layout (Figures 2.3) the stride is 1. If the stride is less than the cache line size, consecutive memory references in the innermost loop are likely to access the same cache line, therefore increasing the likelihood that a memory access is satisfied in the cache.

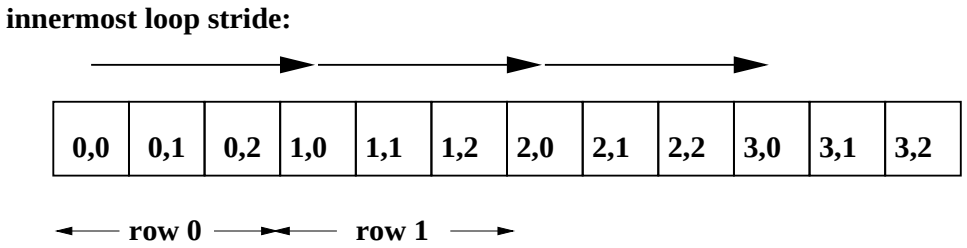

Figure 2.2: Row-major layout

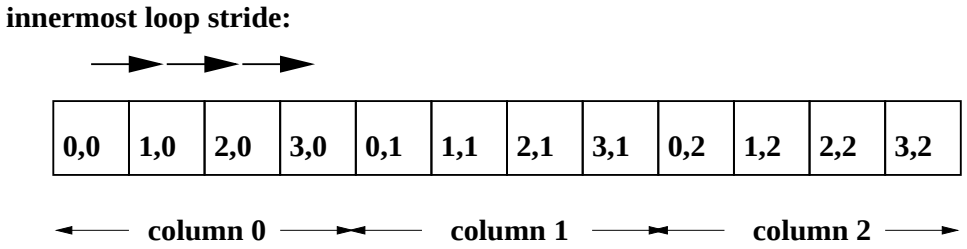

Figure 2.3: Column-major layout

```
real A[n, n];
for i = 1, n/2
  for j = 1, n/2
    R[i, i+j] = ...
```

Figure 2.4: Array reference example

Consider the array reference in the example shown in Figure 2.4. The indexing function is defined by an affine mapping and can be used to compute the subscripts vector:

$$S = Au + \delta = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} i \\ i+j \end{pmatrix}$$

We will use the linear part of the indexing function. The constant part of the affine mapping can be ignored if we do not consider group reuse. Therefore expressions in subscripts of an array reference can be described by an *access matrix* [45]. In our example:

$$A = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

Let u be a vector of loop variables. For our example $u = \begin{pmatrix} i \\ j \end{pmatrix}$. With this

notation, the subscript vector can be computed as: $S = Au$. For our example:

$$S = Au = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ i+j \end{pmatrix}$$

We have already seen that the subscript vector can be used to compute the offset for an array reference $\zeta = S^T m$.

If we assume row major mapping for our example, the offset is

$$\zeta = S^T m_r = (i, \ \ i+j) \begin{pmatrix} n \\ 1 \end{pmatrix} = (n+1)i + j$$

The *stride* is the difference between offsets for this reference in two subsequent iterations (*i.e.* the stride for loop i assumes instances of the reference with the loop variable of i incremented by the step size and all other loop variables constant).

In our case, the stride in the inner loop is 1 and the stride in the outer loop is $(n+1)$. We can represent this information more concisely with the *stride vector*. In our case the stride vector is $v_r = \begin{pmatrix} n+1 \\ 1 \end{pmatrix}$.

We can compute the stride vector from the mapping vector and the access matrix: $v = A^T m$. Let us consider $m_c = (1, \ n)^T$ and $m_r = (n, \ 1)^T$ which represent column major and row major mappings respectively.

$$v_c = A^T m_c = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ n \end{pmatrix} = \begin{pmatrix} n+1 \\ n \end{pmatrix}$$

$$v_r = A^T m_r = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} n \\ 1 \end{pmatrix} = \begin{pmatrix} n+1 \\ 1 \end{pmatrix}$$

The formula for the offset can be rewritten as:

$$\zeta = (Au)^T m = u^T A^T m = u^T v$$

2.4.3 Optimizations

Desired stride vectors

Let us examine our two stride vectors v_r and v_c defined earlier. None of them shows temporal reuse, but row major mapping will cause spatial reuse in the inner loop and column major mapping will have no reuse. To exploit spatial locality, we want the stride 1 in the inner loop. Therefore, row major mapping is preferable for this array reference and this loop nest.

We consider separately optimal stride vectors for uniprocessor and parallel programs. Let v be a stride vector:

$$v = \begin{pmatrix} v_1 \\ v_2 \\ \cdots \\ v_N \end{pmatrix}$$

where N is the depth of the loop nest enclosing this array reference.

Optimizing for sequential programs

For the best locality, inner loops should have better data reuse than outer loops. This increases the likelihood of cache hits. So, a stride vector, v, is optimal if:

$$(\forall i = 1, \ldots, N-1) \;\; v_i \geq v_{i+1} \tag{2.1}$$

Optimizing for parallel programs

For parallel programs the optimality condition is different. Good locality for parallel programs means good data reuse on one processor and little false sharing between processors. In terms of stride vectors, we want all elements that correspond to sequential loops to satisfy the condition from Equation 2.1 and at the same time each of those elements should be less than any element corresponding to a parallel loop. If possible, strides in all parallel loops should be greater than the coherency unit for a given machine to avoid false sharing.

To simplify the notation, we assume here that in each loop nest all parallel loops are the outermost loops, so that the conditions for parallel and sequential programs are the same. Note that the more general case can be handled by small modifications of the algorithms presented in the following sections.

2.4.4 Application of stride vectors

Stride vectors can be used to decide which data mappings and loop transformations should be used to optimize data reuse. Section 2.5 presents an algorithm for deciding optimal mappings for arrays if only data transformations are performed. Later, Section 3.2 shows how to apply both control and data transformations to improve data locality.

2.5 Data Transformations

In this section we will show how the best mapping for an array can be chosen if we apply data transformations only.

2.5.1 Single reference per array

Consider the simple example from Figure 2.4. The access matrix is $A = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$. We want to decide the mapping for the array `R`. By Theorem 2.2, we know that in this case there are only two choices for linear mappings: $m_r = (n,\ 1)^T$ for row major and $m_c = (1,\ n)^T$ for column major.

As shown in Section 2.4.2, we can now compute stride vectors for both mappings.

$$v_r = A^T m_r = \begin{pmatrix} n+1 \\ 1 \end{pmatrix}$$

$$v_c = A^T m_c = \begin{pmatrix} n+1 \\ n \end{pmatrix}$$

From the stride vectors we see that row major mapping is preferable for this array reference and this loop nest.

As discussed in Section 2.3.2, there are $d!$ different dense mappings, where d is the number of dimensions of the array. Factorial is a fast growing function, but we have rarely seen an array in a real-life application with more than 5 dimensions. The vast majority of arrays used in scientific applications have less than 3 dimensions. An extensive study of Fortran applications performed by Shen *et al.* [60], showed that out of 18,549 array references in their application suite only 107 are to arrays with more than 3 dimensions. Therefore the brute-force algorithm that tries all possible dense, linear mappings and uses the optimality condition from Section 2.4.3 to find the mapping that results in the best stride vector is fast in practice.

2.5.2 Multiple references per array

The problem may become significantly harder if there are many references for one array. It is possible that different references would require conflicting mappings for optimum data locality. In that case, we must use some strategy to resolve the conflict. We can for instance,

- Choose the default mapping in case of any conflict. This is a simple way of ensuring that the proposed data mapping will always be at least as good as the default mapping.

- Give priorities to different references of the same array and satisfy the locality requirements for the highest priority reference first. Section 6.4.7 describes a conflict-resolution heuristic which works well in practice.

2.6 Experiments

In this section, we first describe the experiment methodology and the applications used. Then we present the results and analysis.

2.6.1 Methodology

We performed three types of experiments.

- **Native timings:** Running the application on the desired computer is the ultimate test of the optimizations proposed here.
- **Hit (miss) ratios:** We use *simulation* and *instrumentation* to find the fraction of memory accesses that were satisfied in the cache. We show that our optimizations increase hit ratio (and, of course, decrease miss ratio).
- **Reference distance:** We instrument the programs to measure reference distances for all array accesses.

In our experiments we used an SGI Challenge. This machine has local caches (1MB) on each node. Hardware supports cache coherence. We present execution times as the results. To obtain hit ratios for uniprocessor executions, we use instrumentation for efficiency—the performance impact of the instrumentation is significant but not as large as the impact of the detailed simulation. We also use a simulator built with Mint [66] to obtain classification of misses as false sharing and other misses for parallel executions. Mint is a relatively fast simulator, but the level of detail causes a larger slowdown than the instrumentation. Instrumentation inserts a function call after every array reference. The instrumentation code uses the address of this reference to update instrumentation information and perform an on-line cache simulation.

In our experiments simulation was more accurate than instrumentation because it used the actual binary representation of the application. The instrumention on the other hand was applied to the source program, and so the optimization introduced by the back-end of the compiler were not taken into effect.

Note that because of the overhead of instrumentation and simulation, the data sizes used in our simulations were smaller than data used in the native runs. The total size of the data in each simulated run was smaller than the actual cache in SGI Challenge. Therefore, to show differences resulting from better locality of a program, we had to simulate machines with smaller caches than the physical configuration of our hardware.

2.6.2 Application suite

We have used a set of five explicitly parallel programs: matrix multiply (MxM), transitive closure of a graph (closure), matrix inversion (inversion), Gaussian elimination (Gauss), and Warshall-Floyd all-pairs shortest paths algorithm (Floyd).

2.6.3 Execution times

For explicitly parallel programs we can only apply data transformations. The code cannot be changed because the programmer has already decided on the parallelization and inserted synchronization primitives that prevent the compiler from using loop transformations. All five explicitly parallel kernels used in our experiments have this property. Our graphs show the performance of the original program and of the program with data transformations.

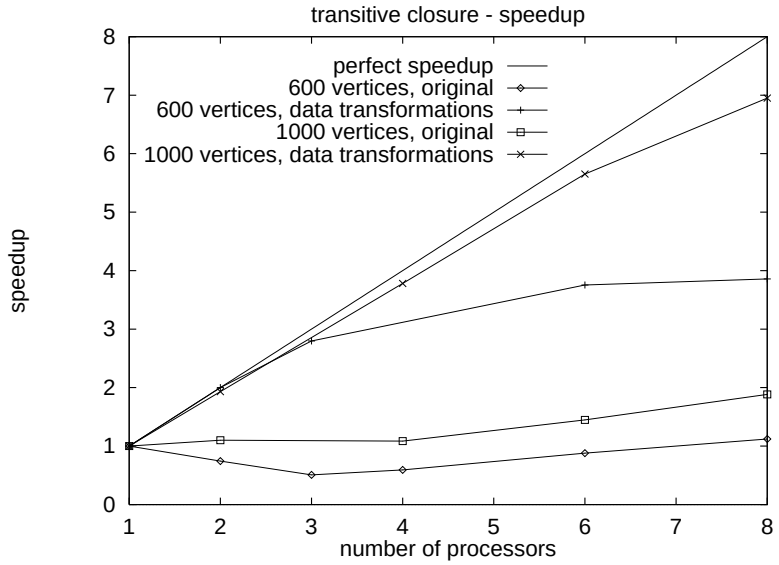

Figure 2.5: Transitive closure: effect of data transformations

Figure 2.5 shows speedups of one of those kernels—transitive closure for two data sizes: for graphs of 600 and 1000 vertices. As is expected, data size must be large enough to provide sufficient parallelism on larger numbers of processors. For this program speedups on up to three processors are good for the optimized version, but when more processors are used, the executions for 1000 vertices show better speedups than executions for 600 vertices.

Similar performance gains were observed for other explicitly parallel applications. Figure 2.6 shows the execution time of the optimized code as a fraction of the execution time of the original program. The applications are a matrix multiply of two 1024×1024 matrices (MxM), shortest paths calculation of a graph

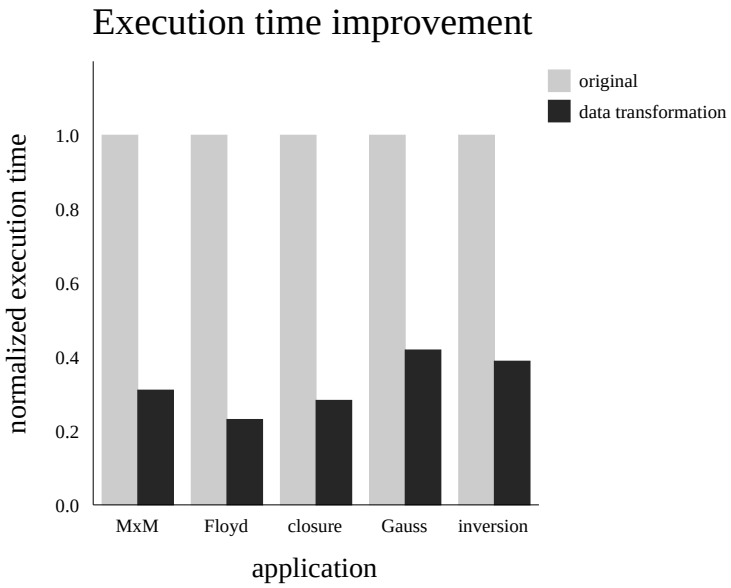

Figure 2.6: Improvements in execution times for selected applications.

with 512 vertices (Floyd), transitive closure of a graph with 350 vertices (closure), Gaussian elimination for 400 equations (Gauss) and matrix inversion of a 500×500 matrix (inversion). All applications are parallel programs and were executed on 8 processors. Instrumentation results discussed in Section 2.6.4 shows that most of the improvement in this case comes from eliminating false sharing.

2.6.4 Cache hit ratios

Figure 2.7 shows the breakdown of memory accesses in the transitive closure program on 2 and 4 processors. For the original program (the left bar in each group) the number of misses and specifically false sharing misses increases as the number of processors increases. For both numbers of processors, data transformations almost completely eliminate misses. Figure 2.5 shows that the performance impact is very dramatic.

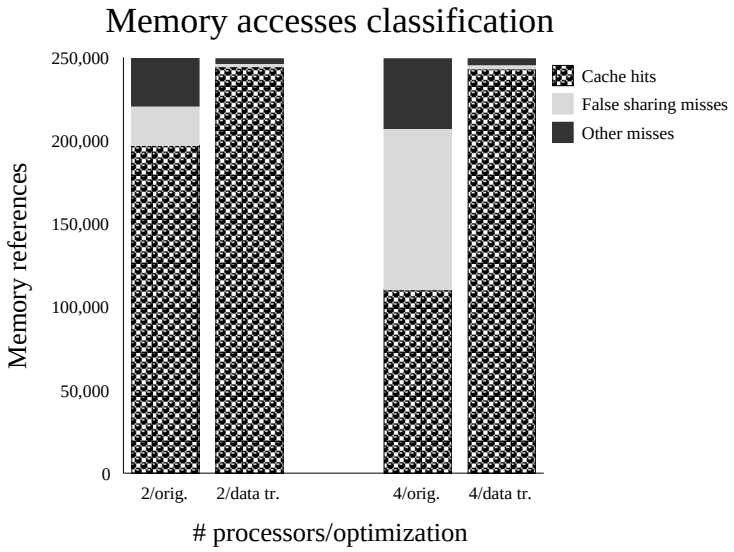

Figure 2.7: Transitive closure: classification of memory accesses

3 Unifying Array and Loop Transformations

3.1 Motivation

Loop transformations (described in Section 2.2.1) and array transformations (Section 2.2.2) have some advantages and disadvantages.

Section 2.2.2 has shown advantages of data transformation over loop transformations. Here is why loop transformations are preferred in some cases.

- Loop transformations are well understood. Substantial work has been done to develop loop transformations for locality and some of the algorithms have been implemented successfully.
- They may improve temporal locality. Data transformations cannot achieve that.
- They work in some cases in which data transformations cannot be applied because of conflicts between different references to the same array. As an example, consider the code fragment created by concatenating loop nests from Figure 2.1a and Figure 2.1b. The first loop nest requires row major and the second one column major mapping. We can however improve locality by performing loop interchange in one of the loop nests.

For some programs, neither data nor code transformations alone will result in the best possible data reuse. We will demonstrate this on the example shown in Figure 3.1. This program computes a product of three square matrices: $F = EAB$.

Assume default column major mapping (similar reasoning may be used for row major mapping). There are three possible approaches to optimizing this program. We will consider all of them to show that for this example, we need both data and control transformations.

```
for i = 1, n
  for j = 1, n
    for k = 1, n
      C[i, j] = C[i, j] + A[i, k] * B[k,j]
    endfor
  endfor
endfor

for i = 1, n
  for j = 1, n
    for k = 1, n
      F[i, j] = F[i, j] + E[i, k] * C[k, j]
    endfor
  endfor
endfor
```

Figure 3.1: MxMxM (multiplication of three matrices): original code

- **Data transformations:** If we do not apply loop transformations, the algorithm from Section 2.2.2 will show that the best locality is achieved when arrays `A` and `E` have row major organization, arrays `B` and `F` have column major organization, and the array `C` cannot have optimal locality in both loop nests (the optimal mapping for the first loop nest is row major and for the second loop nest it is column major).

- **Loop transformations:** If we assume that all arrays have column major organization, the best locality can be achieved if we apply loop interchange in both loop nests to make the i loop innermost [47].

- **Data and Loop Transformations:** We can use the approach presented in this chapter to generate code with even better locality. The code is shown in Figure 3.2 and the following data mappings must be used: `C`, `B` and `F`—column major; `A` and `E`—row major.

  Note that this result cannot be obtained by applying data and loop transformations one after another (in any order).

From the above discussion, we conclude that applying control and data transformations separately will *not* give the best program. A unified approach that utilizes control and data transformations at the same time becomes necessary.

```
for j = 1, n
  for i = 1, n
    for k = 1, n
      C[i, j] = C[i, j] + A[i, k] * B[k,j]
    endfor
  endfor
endfor

for j = 1, n
  for i = 1, n
    for k = 1, n
      F[i, j] = F[i, j] + E[i, k] * C[k, j]
    endfor
  endfor
endfor
```

Figure 3.2: MxMxM: control and data transformations

3.2 The Unified Approach

In this section we will show how to unify loop transformations with the data transformation approach presented in Section 2.5.

We present the framework for representing control and data transformations, which can be used to design algorithms for deciding the combination of loop transformation and data mappings that results in good data locality. It is hard to find an optimal solution in this case and therefore the algorithm presented here is a heuristic. However, in most common patterns of code, the generated code will possess the optimal data locality.

3.2.1 Linear Loop Transformations

The control transformation we use here is the linear loop transformation theory developed in [46]. This framework makes it possible to employ complex program transformations to improve data locality and eliminate false sharing for banded matrix programs. This transformation theory is based on the use of integer lattices as the model of loop nests and the use of non-singular matrices as the model of loop transformations.

Consider two nested loops. The points in the iteration space of this loop can be modeled as integer vectors in the two dimensional space $\mathbb{Z}^2$, where $\mathbb{Z}$ is the set of integers. For example, the iteration ($i = 2$, $j = 3$) can be represented by the vector (2, 3). In general, points in the iteration space of a loop nest of depth n can be represented by integer vectors from the space $\mathbb{Z}^n$.

We will focus on transformations that can be represented by linear, one-to-one mappings from the iteration space of the source program to the iteration space of the target program. Linear, one-to-one mappings between iteration spaces can be modeled using *integer, non-singular matrices*. Performing a sequence of transformations corresponds to composing the mappings between iteration spaces, which, in turn, can be modeled as the product of the matrices representing the individual transformations. The issues of code generation and legality testing are described in [46]. The framework has been used for reducing non-local memory accesses on NUMA parallel machines [45].

3.2.2 Single array reference

To capture the data locality for array accesses, we will use the concept of a stride vector introduced in Section 2.4.2. To compute the stride vector in the generated code, we have to apply the loop transformation to the array reference. Mathematically, we have to multiply the transformation matrix T by the access matrix A. Let us consider again the example from Figure 2.4. Assume that we want to apply loop interchange to obtain the code shown in Figure 3.3.

```
real A[n, n];
for J = 1, n/2
  for I = 1, n/2
    R[I, I+J] = ...
```

Figure 3.3: Array reference example after loop interchange

Loop interchange for this loop nest is represented by the matrix: $T = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$.

Recall that the stride vector in the original code can be computed as $A^T m$ where m is the mapping vector for array `R`.

We can compute the subscript vector in terms of the new loop indices $S = AT^{-1}u$ where u contains loop indices of the transformed loop nest. For our ex-

ample,

$$S = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} J \\ I \end{pmatrix} = \begin{pmatrix} I \\ I + J \end{pmatrix}$$

The offset for an array reference in the transformed code is

$$\zeta = S^T m = (AT^{-1}u)^T m = u^T (T^{-1})^T A^T m$$

If all steps are unitary: $\zeta = u^T v$ (see Section 2.4.2).

Hence, the stride vector is $v = (T^{-1})^T A^T m$. For our example and row major mapping,

$$v_r = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} n \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ n + 1 \end{pmatrix}$$

To verify the correctness, we can compute the new stride vector directly. The access matrix in the code in Figure 3.3 is $A_x = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. Hence, the stride vector for row major mapping is

$$v_r = A_x^T m_r = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} n \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ n + 1 \end{pmatrix}$$

which matches the previous result.

We want to find a loop transformation and an array layout that will result in the best data locality. More formally, we want to find a transformation matrix T and a mapping vector m that will result in a stride vector v with elements in decreasing order. At the same time the transformation matrix must satisfy data dependences and the mapping vector must represent a legal mapping.

We use nonsingular transformation matrices, therefore, we can rewrite $v = (T^{-1})^T A^T m$ as,

$$T^T v = A^T m$$

In principle, we could find the optimal solution by solving this equation (with the constraints for legality of transformation and mapping). This optimization is hard to solve.

We propose a heuristic that is sufficient for many common patterns of code. The search space of possible loop transformations is potentially infinite. In our algorithm, we assume that the transformation matrix contains only values 0 and 1. We also assume that we know the value of the stride vector v beforehand. We construct the matrix T row by row by trying all possible values for this row and all dense, linear mappings. We verify if the rows constructed up to the current step satisfy the above equation and the data dependences.

This idea can be realized in a two-step algorithm. In the first step, we find the desired stride vector in the target code for every array reference. Note that

because of the relation between different references to the same array, not all references will have the ideal stride vector as described in Section 2.4.3. In the second step, we construct the matrix T^T row by row.

3.2.3 Multiple array references

A similar algorithm can be used to solve the most general problem of multiple array references in multiple loop nests. The search space is larger in this case than for the single array reference. If there are many arrays, we initially consider all possible mappings for every array. As the transformation matrices are being built, we constrain the set of acceptable mappings for every array.

As in Section 2.5.2, the unified algorithm for multiple array references must implement conflict resolution.

3.3 Experiments

For the experiments we have used the methodology described in Section 2.6.1.

3.3.1 Application suite

We have used a set of three sequential Fortran programs which are automatically parallelized. The code shown in Figure 3.4 is a synthetic benchmark that can benefit from unified data locality improvements.

```
for i = 1, n
  for j = 1, n
    A[i, j] = B[j, i]*C[i, j] + D[i, j]*LOG(E[j, i])
  endfor
endfor

for i = 1, n
  for j = 1, n
    B[i, j] = A[j, i] + E[i, j]
  endfor
endfor
```

Figure 3.4: A synthetic benchmark

The second program, *MxMxM*, is the example from Figure 3.1 which was discussed in Section 3.1. This program computes the product of three square matrices.

The third benchmark—*economics*—is a larger application from the domain of economics modeling. It calculates the amount of goods shipped between supply and demand markets given supply and demand prices, transportation costs and tariffs [55].

3.3.2 Execution times

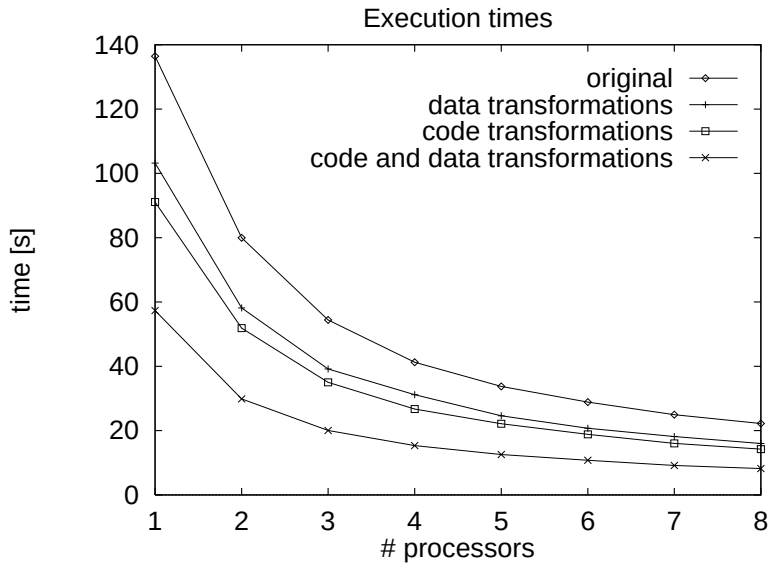

Figure 3.5: Execution times for the synthetic benchmark

Execution times for the synthetic benchmark are shown in Figure 3.5. We can see that both data transformations and loop transformations alone improve performance of the program, but unified optimizations result in shorter execution times than either of those transformations alone. Unified transformations decrease the execution time by about 40% as compared to loop transformations only.

Figure 3.6 shows that the unified set of optimizations produces not only the best performance for the uniprocessor case, but also the most scalable parallel program. Each speedup is relative to the optimized version of the sequential program, rather than the original unoptimized version. Therefore speedups for one processor are always 1 for all four versions of the program, even though Figure 3.5 demonstrates that the uniprocessors times of the optimized versions are different.

The effect of scheduling (assigning iterations to processors) was also eliminated as much as possible. The best possible scheduling was used in the unoptimized case

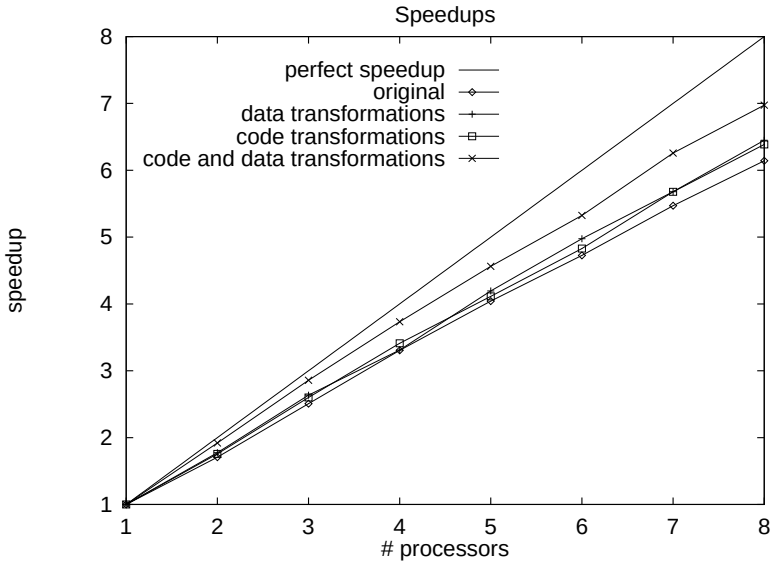

Figure 3.6: Speedups for the synthetic benchmark

so that there is no additional improvement in the optimized case due to scheduling. Parallel programs whose timings are presented in Figures 3.5 and 3.6 had their parallel loops blocked to minimize false sharing. If a naive scheduling algorithm, e.g. interleaving, were used in the original version, locality optimizations would have a bigger impact on the program behavior and the relative improvements in performance would be larger.

We can see that the version with unified optimizations scales better than any of the other three implementations. Even the uniprocessor execution benefits from our approach, but the improvement increases with the number of processors.

For the double matrix multiply, we have applied all three possibilities of locality optimizations: data transformation alone, loop transformation alone and the unified transformation (see the discussion in Section 3.1). We can see in Figure 3.7 that for this program we have to apply both data and control transformations to obtain the shortest execution time (17% faster than loop transformations only). Again, the optimizations help for both uniprocessor and parallel executions.

Figure 3.8 shows the execution times for the economics program. We can see that applying transformations cuts the execution time by more than half. Applying both data and code transformations produced the code 8% faster than the code with loop transformations only.

3.3.3 Cache hit ratios

Figure 3.9 shows numbers of misses for MxMxM for different cache sizes. We assume a direct-mapped cache. To make the graph more readable, miss numbers

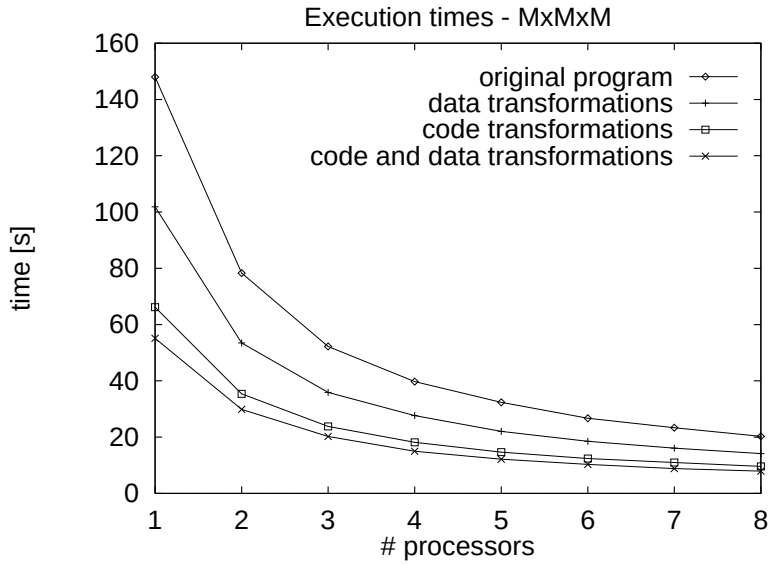

Figure 3.7: Execution times for the MxMxM program

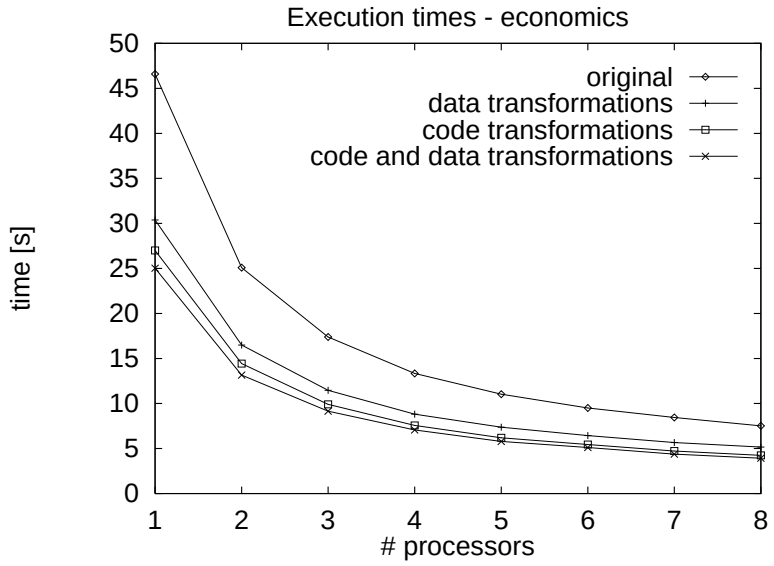

Figure 3.8: Execution times for the economics program

are normalized for each cache size so that the unoptimized code had one unit of cache misses. As discussed in Section 2.6.1 the instrumentation incurs a significant overhead in terms of the execution time. Therefore, we had to restrict ourselves to smaller data sizes. Consequently, we had to simulate caches with smaller sizes than the actual hardware present in our machines. We varied the cache size from 16 Kbytes to 128Kbytes, the cache line size was 32 bytes.

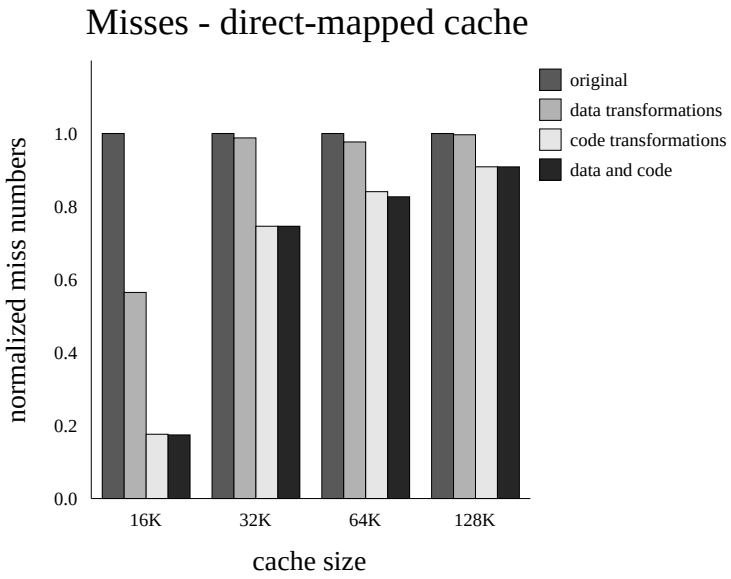

Figure 3.9: Normalized miss numbers for MxMxM

3.3.4 Reference distances

The experiments presented in this section show that the optimizations discussed in this chapter decrease reference distances for many array references. Reference distance is a useful metric, because it is not very closely tied to machine parameters and yet it can be used to successfully predict (*qualitatively* rather than *quantitatively*) the impact of locality optimizations. The cache line size in all experiments in this section was 32 bytes.

Figure 3.10 presents numbers of references whose reference distance falls into a given interval. We have divided the distances into intervals that best show the effect of data transformations on the economics program. We can see that the transformations increased the number of references in the 5–10 interval and decreased the number of references with larger distances—in the 101–500 interval. References with other distances remained basically unchanged.

Larger reference distance means higher probability that the cache line has been evicted since its last use. Recall that because of the significant impact of the instrumentation, we had to run the program on small data sets which explains why all reference distances are small.

Figure 3.11 shows reference distance intervals for the MxMxM program from Figure 3.1 and its three optimized versions which were described in Section 3.1. This is presented in a slightly different way than for the economics program in Figure 3.10, because we compare four versions of the program instead of two.

In the optimized code, the number of references in intervals corresponding to

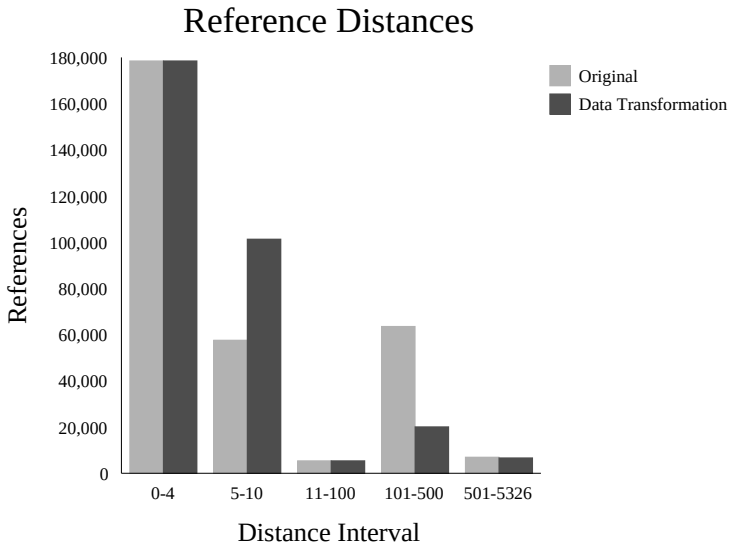

Figure 3.10: Reference distances for the economics program.

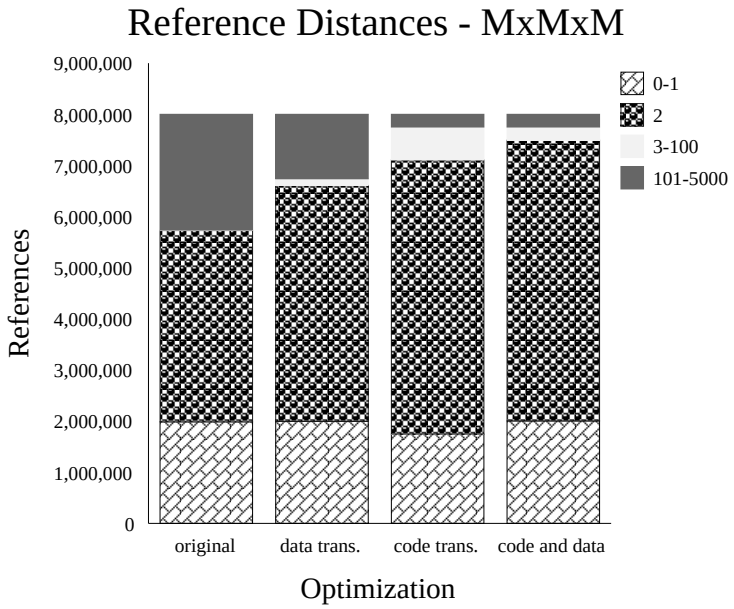

Figure 3.11: Reference distances for MxMxM.

small distances increases and at the same time the number of references with large distances decreases (the total number of references does not change). The shifts of reference distances correlate well with decreases in execution times presented in Figure 3.7.

4 Data Structure Recovery

4.1 Motivation

Understanding the pattern of array accesses has been identified as a key element of many optimization techniques: it is necessary for achieving good locality of memory references, for deciding data distribution, and for determining data dependences.

For *distributed shared-memory machines* and uniprocessor machines, we have shown in Chapters 2 and 3 the need to optimize the programs for locality by changing the array layout in memory. Experiments have shown that unified data and loop transformations may significantly improve performance of some applications. However, data transformations are impossible without the multidimensional array structures.

The importance of the right array layout is even greater for NUMA (non-uniform memory access) parallel machines. Compilers for modern NUMA machines like Origin2000 provide mechanisms for specifying array layouts [15, 61].

For *distributed-memory message-passing machines*, a parallelizing compiler must decide the placement of data into the local memories of participating processing units. For every array, the compiler must compute a function mapping from data space to a processor space [43, 32, 16, 3, 7]. Such algorithms also depend on the multiple dimensionality of arrays.

Unfortunately, in many real applications arrays do not have interesting structures, i.e., they are plain linear arrays rather than multi-dimensional arrays. For example, consider the code in Figure 4.1 from the subroutine OLDA in the program TRFD in the PERFECT benchmark suite [8]. This array structure leaves little room for powerful compiler optimizations based on data structures. Logically, however, the array `XIJRS` has a nice 4-dimensional structure which can be exploited in a compiler.

Array linearization, which generates linear array structure and accesses, is frequently used in scientific programs [48, 53]. Maslov [53] reports that 6 out of

```
      m=0
      DO 300 i=1,n
        DO 300 j=1,i
           ...
           DO 240 r=1,n
             DO 230 s=1,r
               m=m+1
               VAL=XIJRS(m)
               ...
230          CONTINUE
240        CONTINUE
           ...
300   CONTINUE
```

Figure 4.1: A loop nest from TRFD/OLDA

8 programs in the RiCEPS [12] benchmark suite contain linearized references. In addition to the applications from RiCEPS, we have studied programs from the PERFECT Benchmarks [8] suite. The study shows that linearized array references are common in PERFECT applications.

We show that interesting logical data structures can be automatically recovered from the flat one-dimensional arrays and accesses. We present a framework and algorithms for recovering the logical multidimensional data structures.

There has been very little work on recovering logical structures of arrays. The closest is the work by Maslov [53], where he developed a data dependence algorithm capable of detecting data dependences in the complicated access patterns to the linear arrays. The dependence analysis for linearized references has been further refined in Maslov and Pugh [54].

In the rest of this chapter, we first present the conceptual framework for our algorithm. Later, in Sections 4.3 and 4.4 we present algorithms for recovering array structures for rectangular and non-rectangular arrays, respectively. Section 4.5 shows how to resolve possible conflicts for multiple references to the same array.

4.2 Framework

4.2.1 Notation

To discover the number of dimensions and recovered subscript expressions, we use the concepts introduced in Section 2.3. A simple example in Figure 4.2 will be used as an illustration.

```
REAL A(0:199, 0:99)
DO i = 1, 99
  DO j = 1, 100
    DO k = 0, 99
      A(199 - 2 * k, i - 1) = ...
    END DO
  END DO
END DO
```

Figure 4.2: A 2-D array reference

Let u be a vector of loop variables. For our example $u = \begin{pmatrix} i \\ j \\ k \end{pmatrix}$. With this notation, the subscript vector can be computed as: $S = Au + \delta$. For our example:

$$S = Au + \delta = \begin{pmatrix} 0 & 0 & -2 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 199 \\ -1 \end{pmatrix} = \begin{pmatrix} 199 - 2k \\ i - 1 \end{pmatrix} \ .$$

Here A is the access matrix. The mapping vector for this array is $m = \begin{pmatrix} 1 \\ 200 \end{pmatrix}$ and it can be interpreted to mean that a single element is of size 1 and a single column has the size 200.

For notational convenience, we define $\overline{S} = Au$, so that $S = \overline{S} + \delta$.

For a given reference, the mapping vector and the subscript range vector are closely related. The following is true:

$$(\forall i = 2, \ldots, d) m_i = \prod_{j=1}^{i-1} w_j \ . \tag{4.1}$$

Whenever necessary (mainly for code generation phase), we will include one additional element in a mapping vector representing the correction necessary to make the offset for the first element of the array equal to zero. In that case, we will also add an additional element to the subscript vector, $S_{d+1} = 1$. For the example from Figure 4.2, the extended vectors would be: $S = \begin{pmatrix} 199 - 2k \\ i - 1 \\ 1 \end{pmatrix}$, $m = \begin{pmatrix} 1 \\ 200 \\ 0 \end{pmatrix}$, so that the offset is $\zeta = S^T m = 200i - 2k - 1$. Unless explicitly stated otherwise, we use simple mapping vectors, not the extended vectors.

Many of our algorithms are based on the following simple equality from Section 2.4.2: $A^T m = v$.

Arrays with constant mapping vectors are called *rectangular*. This is the only kind of an array supported by type systems of programming languages like C or Fortran. We extend the notion of an array to multidimensional structures which can be represented with variable mapping vectors. We call those arrays *non-rectangular*.

4.2.2 A Legal Fortran Mapping

For rectangular arrays, we assume that recovered mappings must correspond to legal arrays in an existing programming language. Without loss of generality, we assume that we want to recover Fortran (i.e., column major) arrays whose subscripts start from 0. Note that we do not have enough information in the linearized reference to recover lower bounds of subscript ranges. We can only discover how many elements there are in each array dimension.

A mapping vector $m = \begin{pmatrix} m_1 \\ \vdots \\ m_d \end{pmatrix}$ represents a legal Fortran mapping if the following conditions are satisfied:

1. $m_1 = 1$,

2. $(\forall i = 2, \ldots, d) m_i \geq m_{i-1}$, and

3. $(\forall i = 2, \ldots, d) m_i \mod m_{i-1} = 0$.

4.3 Rectangular Arrays

4.3.1 An Overview

We concentrate on recovering structures of affine array references. We assume that we start with a one-dimensional, affine array reference enclosed by a loop nest and we want to turn this reference into a multidimensional reference. With those assumptions, we describe a multidimensional array reference with a mapping vector, an access matrix, and an offset vector (see Section 4.2.1).

We can recover this structure in two steps:
1. First we compute the stride vector and use it to find a mapping vector and an access matrix.
2. Then we complete the subscript expressions by finding an offset vector.
We present solutions to those two steps in the following sections.

If there is more than one reference to an array in the analyzed portion of the code, we may recover different multidimensional structures for the same array. We show later how to merge those different array definitions into one consistent array type.

```
REAL A(0:19999)
DO i = 1, 99
  DO j = 1, 100
    DO k = 0, 99
      A(200i - 2k - 1) = ...
    END DO
  END DO
END DO
```

Figure 4.3: A linearized reference

Throughout this section we will use the loop nest from Figure 4.3 to illustrate the algorithm for recovering a multidimensional structure of an array. After each step of the algorithm, we will show how that step is applied in practice.

4.3.2 Computing a Mapping Vector and an Access Matrix

Finding a Stride Vector

Given a one-dimensional subscript expression b and a loop nest structure, we can easily find a stride vector for an array reference by calculating the difference between values of b for subsequent iterations.

The stride vector need not be constant. Nevertheless, in practice, strides tend to be either constant or simple functions of loop variables. In some cases they may be more complex, e.g., as a result of indirection. In this section, we assume that strides are loop-invariant (more general stride vectors are discussed later). If the stride vector is constant, the recovered array is rectangular and it can be represented as a built-in array type in programming languages like C or Fortran.

Example: Consider the loop nest shown in Figure 4.3. The subscript expression is $b = 200i - 2k - 1$. Since $\left(b\Big|_{i=x+1} - b\Big|_{i=x}\right) = 200$, the stride for loop i is 200. Computing similarly strides for the other two loops, we get the stride vector:

$$v = \begin{pmatrix} 200 \\ 0 \\ -2 \end{pmatrix}.$$

Finding a Mapping Vector

In general, the solutions space for the mapping vector and the access matrix may be infinite and finding a *meaningful solution* may be difficult. We present here an algorithm which will find a solution for linearized references to rectangular arrays. Our solution is meaningful in the sense that the recovered array structure matches its use.

In this approach, we first find a mapping vector and then compute an access matrix which is consistent with this mapping vector and the stride vector calculated earlier.

We observe that in many cases each subscript expression of the recovered logical array reference contains at most one loop variable and for a given reference each loop variable appears in at most one subscript expression. We call this condition the *simple subscripts condition*.

This observation leads us to a conclusion that a mapping vector found as a sorted permutation of non-zero elements of the stride vector will be a legal Fortran mapping [1].

[1] As discussed later, we have to take care of a few details, but this is the basic idea of our approach.

When analyzing a linearized array reference, we do not know whether the recovered logical multidimensional reference satisfied the simple subscripts condition. Therefore, we initially assume that the condition is indeed true and try to apply the algorithm. At various points of the algorithm the validity of the solution is verified. If a solution is determined to be invalid, the algorithm fails.

The algorithm presented here assumes that in a given loop nest all non-zero strides are different. We start with computing the stride vector v. We verify that v is constant. Now, we create a vector v' by removing from v all elements equal to 0 and ordering all non-zero elements by their absolute values. Number of elements of v' is the number of dimensions, d, in the recovered array structure. The mapping vector m is created as an element-wise absolute value of v'. If the absolute value of the smallest element of v' is greater than 1, we make this element equal to one.

After computing the mapping vector, we verify that it defines a legal Fortran mapping, as explained in Section 4.2.2. If it does, we can proceed to finding an access matrix.

Example: Compressing and sorting the stride vector $v = \begin{pmatrix} 200 \\ 0 \\ -2 \end{pmatrix}$ produces: $v' = \begin{pmatrix} -2 \\ 200 \end{pmatrix}$. And the mapping vector is: $m = \begin{pmatrix} 1 \\ 200 \end{pmatrix}$.

Finding an Access Matrix

First, we compute the subscript range vector, w, using (4.1). We need w to represent the array, but it is also used to resolve potential ambiguities while building the access matrix. Note that from (4.1) we can compute all but the last element of the subscript range vector, i.e., w_d. The last element is not required for uniprocessor code generation (unless we want to check subscript ranges at run-time), but it may be necessary to decide data distribution. We compute it later from the subscript expression in the last dimension and the ranges of loop variables involved in that expression.

Given v and m, we want to find A which satisfies: $A^T m = v$.

Recall that we assume that the multidimensional array reference satisfies the simple subscripts condition (defined in Section 4.3.2). In that case we can construct the access matrix column by column.

Consider column i. We have:

$$v_i = \sum_{j=1}^{d} A_{j,i} m_j \quad .$$

There are two cases:

1. $|v_i| = m_j$ for some j. In this case we make $A_{j,i} = \frac{v_i}{m_j}$ and $A_{k,i} = 0$ for all $k \neq j$. Note that the value of $\frac{v_i}{m_j}$ is either 1 or -1 depending on the sign of v_i.

2. Otherwise we choose the maximum m_j such that $m_j < |v_i|$ (given the algorithm described in Section 4.3.2 for finding the mapping vector, we know that $j = 1$). We make $A_{j,i} = \frac{v_i}{m_j}$ and $A_{k,i} = 0$ for all $k \neq j$.

After finding the access matrix, the only information needed to generate the multidimensional reference is the offset vector. We show how to find it in Section 4.3.3.

Example: For $v = \begin{pmatrix} 200 \\ 0 \\ -2 \end{pmatrix}$ and $m = \begin{pmatrix} 1 \\ 200 \end{pmatrix}$, we can find the subscript range vector: $w = \begin{pmatrix} 200 \\ ? \end{pmatrix}$ The algorithm described in this section builds the following access matrix: $A = \begin{pmatrix} 0 & 0 & -2 \\ 1 & 0 & 0 \end{pmatrix}$. The subscripts of the reference are: $S = \begin{pmatrix} -2k + \delta_1 \\ i + \delta_2 \end{pmatrix}$. We will show how to compute δ_1 and δ_2 in the next section.

4.3.3 Computing an Offset Vector

In Section 4.3.2 we have recovered the structure of an array, i.e., the mapping vector m and the parts of subscripts expressions which depend on loop variables, i.e., the access matrix A. Now, we have to compute constant parts of the subscripts expressions.

More formally, we want to find the vector δ as defined in Section 4.2.1. The vector δ is subject to two constraints:

1. All elements of the subscript vector $S = Au + \delta$ must be within the array bounds defined by the subscript range vector w. Recall that subscripts in the recovered array structure start from 0 (compare Section 4.2.2).

2. The value of the offset $\zeta = S^T m$ for u being a zero vector must be equal to the constant part t_1 of the one-dimensional subscript expression b.

The second constraint will be satisfied if $(A\vec{0} + \delta)^T m = t_1$ or simply $\delta^T m = \sum_{k=1}^{d} \delta_k m_k = t_1$. As $m_1 = 1$ and $w_1 = \frac{m_2}{m_1} = m_2$, we can rewrite the equation as:

$$\delta_1 + \overline{\delta}_1 w_1 = t_1 \quad \text{where} \quad \overline{\delta}_1 = \delta_2 + \frac{\sum_{j=3}^{d} \delta_j m_j}{m_2} \tag{4.2}$$

There are infinitely many solutions to (4.2). We first find an arbitrary solution $\delta_1', \overline{\delta}_1'$: $\delta_1' = t_1$ and $\overline{\delta}_1' = 0$.

Then we find the value of δ_1 that satisfies the condition for the range of the subscripts in the first dimension: $0 \leq \overline{S}_1 + \delta_1 < w_1$ by subtracting a multiple of w_1 from δ_1': $\delta_1 = \delta_1' - t_2 w_1$.

The solution to (4.2) can be completed by setting $\overline{\delta}_1 = t_2$. It is easy to verify that the pair $\delta_1, \overline{\delta}_1$ satisfies (4.2).

By substituting the value of $\overline{\delta}_1$ in the formula for $\overline{\delta}_1$ in (4.2) with t_2, we obtain an equation that can be used to compute δ_2:

$$\delta_2 + \overline{\delta}_2 w_2 = t_2 \quad \text{where} \quad \overline{\delta}_2 = \delta_3 + \frac{\sum_{j=4}^{d} \delta_j m_j}{m_3} \tag{4.3}$$

Notice that (4.3) is similar to (4.2). We can generalize this algorithm for any dimension. For dimension i we start with the equation:

$$\delta_i + \overline{\delta}_i w_i = t_i \tag{4.4}$$

where $\overline{\delta}_i = \delta_{i+1} + \frac{\sum_{j=i+2}^{d} \delta_j m_j}{m_{i+1}}$. We find $\delta_i = t_i - t_{i+1} w_i$ which satisfies: $0 \leq \overline{S}_i + \delta_i < w_i$. The value of t_{i+1} will be used to compute δ_{i+1}.

Note that for the last dimension (4.4) is degenerated to: $\delta_d = t_d$ and we get δ_d directly without adding the correction. However, we still have to verify that the condition: $0 \leq \overline{S}_d + \delta_d < w_d$ is satisfied.

Example: We continue recovering structure of the reference from Figure 4.3. We have $t_1 = -1$.

Now we compute the offsets:

- Dimension 1: $\delta_1 + 200\overline{\delta}_1 = -1$. We compute $\delta_1 = -1 - 200 t_2$ such that $0 \leq \overline{S}_1 + \delta_1 < w_1$ where $\overline{S}_1 = -2k$. By computing the values of $\overline{S}_1$ for the lower and upper bound of the loop k, we can determine that we should use $t_2 = -1$. Hence $\delta_1 = 199$.

- Dimension 2: $\delta_2 = t_2 = -1$.

The subscripts vector of the recovered two-dimensional reference is

$$S = \begin{pmatrix} 199 - 2k \\ i - 1 \end{pmatrix}.$$

We can see that the loop nest from Figure 4.3 is equivalent to the loop nest from Figure 4.2.

We are now able to compute the last element of the subscript range vector, w. We have already recovered all but the last elements of w. The subscript expression in the last dimension is $S_2 = i - 1$ and its range is 0..98. Therefore the full vector is: $w = \begin{pmatrix} 200 \\ 99 \end{pmatrix}$ The recovered array: `A(0:199, 0:98)` has one column less than the array declared in Figure 4.2. This is correct as the last column of A is never referenced in this loop nest.

4.4 Non-rectangular Arrays

In this section we will show how to recover a multidimensional structure of a non-rectangular array. We first describe the problem and propose a solution for a general class of arrays whose subscript ranges can be described as functions of the values of other subscripts. Later, in Section 4.4.3, we discuss a special case of non-rectangular arrays: *triangular arrays* which are common in practice.

4.4.1 General arrays

In Section 4.3, we assumed that the stride vector is constant. In this case, if we are able to recover the structure of an array, the array will be rectangular.

```
DO i = 1, n
  DO j = 1, i*i
    A(j + (i*(i*(2*i -3) + 1)) / 6) = ...
  END DO
END DO
```

Figure 4.4: Non-rectangular loop

Consider the reference in Figure 4.4. The stride for loop i is not constant. To see this, let us consider the subscript expression for two consecutive iterations of loop i: $b_i = j + \frac{2i^3 - 3i^2 + i}{6}$ and $b_{i+1} = j + \frac{2(i+1)^3 - 3(i+1)^2 + (i+1)}{6}$. The difference between

those expressions is the stride for loop i: $v_1 = b_{i+1} - b_i = i^2$. The stride vector is: $v = \begin{pmatrix} i^2 \\ 1 \end{pmatrix}$. The techniques developed in Section 4.3 are not directly applicable here.

In this example, although a closer inspection reveals that the logical structure of array `A` seems to have two dimensions, there is no way to convert the reference in Figure 4.4 into a two-dimensional array reference in a classical programming language like Fortran. To make the logical number of dimensions explicit, we need to introduce a new notation. We allow a subscript range to be a function of the values of other subscripts.

So, a two-dimensional array would have a subscript range vector:

$$w = \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} \ ,$$

where F_1 and F_2 are functions mapping a subscript vector to a positive integer, i.e., $F_1, F_2 \colon \mathbb{Z}^d \to \mathbb{Z}$

For the example given in Figure 4.4, we would like to be able to recover a two-dimensional array with the following subscript ranges: $F_1 \equiv \lambda x.\lambda y.y^2$ and $F_2 \equiv \lambda x.\lambda y.n$ (or, equivalently, $F_1(x, y) = y^2$ and $F_2(x, y) = n$). In other words, the number of elements in the first dimension is equal to the square of the value of the subscript in the second dimension, and the number of elements in the second dimension is N.

In principle, functions which determine subscript ranges may be very complex, but in practice we have only encountered the following two functions:

1. A *constant* — this is the simplest case. If subscript ranges for all dimensions are constant, we have a rectangular array (see Section 4.3) which is allowed by existing programming languages.

2. A *linear function* of a subscript expression in another dimension (the subscript range of that dimension is constant). Usually (recall that we assume column major mapping), a linear subscript range depends on the subscript in the *next* dimension, i.e., $F_i(S_1, \ldots, S_d) = S_{i+1}$. Arrays with such subscript ranges are called *triangular*.

For non-rectangular arrays, the compiler must generate code for an array reference in a more general way than for rectangular array. Code generated for a given reference must compute a linear offset from the base address of an array. For rectangular arrays computing the offset is straightforward [17]: $\zeta = S^T m$ (for simplicity of the discussion we ignore constant terms in the subscript expressions). For non-rectangular arrays, we have to replace the simple multiplication with an

operator which computes the sum of subarray sizes. For constant elements, this sum is of course equivalent to the multiplication used for rectangular arrays.

We use the symbol $\otimes$ to represent this new operator. The offset can be now computed as $\zeta = S^T \otimes m$. Because for constant mapping vectors, $\otimes$ is reduced to multiplication, we can use this new notation consistently whenever we want to compute the offset.

We will use the same symbol $\otimes$ for scalar and vector operations. Operation on vectors consists of pairwise applications of "scalar" $\otimes$ to elements of the two vectors and adding the results (being in effect an "inner product").

The semantics of $\otimes$ can be explained as follows. Consider a subscript vector S and a mapping vector m:

$$S = \begin{pmatrix} S_1 \\ \vdots \\ S_d \end{pmatrix}, \quad m = \begin{pmatrix} m_1 \\ \vdots \\ m_d \end{pmatrix} \quad .$$

Let ζ_i denote a contribution to the offset in ith dimension ($\zeta = \sum_{i=1}^{d} \zeta_i$). For dimension 1, we trivially have $\zeta_1 = S_1$ (assuming that an array element has size 1). Consider the contribution in dimension $i > 1$. We compute it as the sum of the values of m_i for all values of the subscript in the ith dimension:

$$\zeta_i = \sum_{k=1}^{S_i - 1} m_i(S_1, \ldots, S_{i-1}, k, S_{i+1}, \ldots, S_d) \quad .$$

Of course, we do not want to generate code which will compute the sum at run-time in a loop. The compiler must be able to symbolically compute the sum at compile-time. For more complicated functions this may be impossible. For all functions which we have encountered, computing the sum of a series is straightforward.

If m_i is a constant, this sum is equivalent to multiplication of S_i by m_i. Therefore, we can use the $\otimes$ operator for rectangular arrays (Section 4.3).

4.4.2 Recovery Algorithm

Recovering the structure of a non-rectangular array is considerably more difficult than for a rectangular array. For references whose subscript expressions in the recovered, multidimensional reference are either constants or loop variables (which is equivalent to satisfying the simple subscript condition with $\delta = \vec{0}$), we apply an approach similar to the one described in Section 4.3. Below is a sketch of such an algorithm.

First, we compute the stride vector (if the linearized subscript expression b is too complex, the algorithm fails). Then, we sort the stride vector by absolute values. The elements of the stride vector are symbolic expressions of loop variables, therefore finding the ordering of the strides is not trivial. For simple functions, we can do it by considering loop bounds. We can get a mapping vector from the sorted stride vector the same way as for rectangular arrays. Directly from the mapping vector, we can get a subscript range vector expressed in terms of loop variables. To describe the structure of the array in a way independent from this loop nest, we must find a subscript range vector expressed in terms of other subscripts. This is straightforward, because all recovered subscript expressions are either constant or loop variables. Note that every loop variable present in the the stride vector must be used in some subscript of the recovered array reference.

After we have obtained the subscripts vector, we check if the ranges of the subscripts are within appropriate ranges from the subscript range vector.

Example: Let us show how the structure of array `A` from Figure 4.4 may be recovered.

First, we compute $v' = \begin{pmatrix} 1 \\ i^2 \end{pmatrix}$, $m' = \begin{pmatrix} 1 \\ i^2 \end{pmatrix}$, and $w' = \begin{pmatrix} i^2 \\ ? \end{pmatrix}$. Now we can find an access matrix, A, which satisfies: $A^T m = v$: $A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. Since we assume $\delta = \vec{0}$, we can compute

$$S = Au = \begin{pmatrix} j \\ i \end{pmatrix} \quad .$$

Now we want to express the mapping vector and the subscript range vector in terms of subscript expressions rather than loop variables, $m = \begin{pmatrix} M_1 \\ M_2 \end{pmatrix}$ and $w = \begin{pmatrix} W_1 \\ W_2 \end{pmatrix}$, where M_1, M_2, W_1, and W_2: $\mathbb{Z}^2 \rightarrow \mathbb{Z}$. It is clear that $M_1(x, y) = 1$. Because $m'_2 = i^2$ and $S_2 = i$, we know that $M_2(x, y) = y^2$. Similarly, we have $W_1(x, y) = y^2$. To compute W_2, we have to look at the bounds for the loop variable in S_2. By examining the bounds of the i loop, we can see that $W_2(x, y) = n$.

In the code generation phase, we can compute the offset as:

$$\zeta = S^T \otimes m = (j \;\; i) \otimes \begin{pmatrix} \lambda x.\lambda y.1 \\ \lambda x.\lambda y.y^2 \end{pmatrix} = j + \sum_{k=1}^{i-1} k^2 \quad ,$$

$$\zeta = j + \frac{2i^3 - 3i^2 + i}{6} \quad .$$

4.4.3 Triangular Arrays

References like the one in Figure 4.4 are of course possible. However, the most complicated non-rectangular array structure we have encountered in practice is triangular.

Recall that an array is a triangular array if all ranges are of the following two types: (1) constant, or (2) linear and depend on the subscript in the next dimension. Using the functional notation introduced in Section 4.4.1, we can write this condition as $F_i(S_1, \ldots, S_d) = S_{i+1}$ and F_{i+1} is constant.

The notation used in Section 4.4.1 is general in that it can be used to represent non-rectangular arrays with subscript ranges defined by arbitrary functions. It is also cumbersome to use and too general for triangular arrays. To make the further discussion easier, let us specialize this notation. We will use the symbol $\bigtriangledown$ to represent a value of the appropriate subscript. The meaning of $\bigtriangledown$ is a little different for a mapping vector and for a subscript range vector.

1. For a mapping vector, $\bigtriangledown$ represents a value of the subscript of *this* dimension.

2. For a subscript range vector, $\bigtriangledown$ represents a value of the subscript in the *next* dimension.

We can have more than one triangular dimensions in one array. As an example consider the reference to array `XIJRS` in the loop 300 of the subroutine OLDA in the TRFD benchmark. The loop nest is shown in Figure 4.1.

After eliminating the induction variable, we get the following subscript expression:

$$b = s + \frac{r(r-1)}{2} + \frac{n(n+1)}{2}\left(j + \frac{i(i-1)}{2} - 1\right) \quad .$$

The stride vector is: $v = \begin{pmatrix} \frac{n(n+1)}{2}i \\ \frac{n(n+1)}{2} \\ r \\ 1 \end{pmatrix}$. Considering the ranges for i and r, we obtain the following mapping vector: $m' = \begin{pmatrix} 1 \\ r \\ \frac{n(n+1)}{2} \\ \frac{n(n+1)}{2}i \end{pmatrix}$. Using the loop-independent notation, we can write: $m = \begin{pmatrix} 1 \\ \bigtriangledown \\ \frac{n(n+1)}{2} \\ \bigtriangledown \end{pmatrix}$. We can easily find the access matrix and

use it to compute the subscript vector: $S = Au = \begin{pmatrix} s \\ r \\ j \\ i \end{pmatrix}$. Therefore, the logical 4-dimensional array type is `XIJRS`$(\bigtriangledown, n, \bigtriangledown, n)$ and the reference is `XIJRS(s,r,j,i)`.

It is difficult to construct the subscript range vector, but we can find a different way of verifying that the subscripts expressions are within bounds which are legal for a particular mapping. Note that the following approach is valid for triangular arrays, but may not be applicable for more general non-rectangular arrays.

The subscript range for the last dimension is always legal. For each dimension $i < d$, we have to check that the maximum value of the subscript expression does not violate the constraint defined by the subarray size in dimension $i + 1$. More formally, we first find the maximum value, $S_i^{\max}$, of the subscript expression by examining loop bounds for loop variables which appear in this expression. For triangular arrays, each expression in the mapping vector is a function of at most one other subscript expression (the expression in this dimension). Consider both cases for checking the subscript range in dimension i:

1. m_i is constant. We have to verify that $S_i^{\max} m_i \leq m'_{i+1}$
2. $m_i \equiv \bigtriangledown$. We have to verify that $\sum_{k=1}^{S_i^{\max}} k m_{i-1} \leq m_{i+1}$ or simply $m_{i-1} \frac{S_i^{\max}(S_i^{\max}+1)}{2} \leq m_{i+1}$. Note that both m_{i-1}, m_{i+1}, and $S_i^{\max}$ are guaranteed to be constant for triangular arrays.

4.5 Multiple References

The algorithms presented in previous sections consider each array reference separately. The drawback of this approach is that it may happen that if there are multiple references to the same array, several different structures may be recovered for the same array. It is desirable to have exactly one type for a given object.

In our framework, we try to unify all structures for a given array, so that the same mapping vector may be used for any reference to the same array. In many cases it is possible. If the types cannot be unified, we are left with two options: we can either use linear structure for all references, or we have to deal with having inconsistent types for the same array.

We have encountered two different code patterns which result in recovering different types. The first—more common—occurs when the loop structure is linearized. The second case results if the dimensionality of an array is unknown at compile time. In the rest of this section, we give examples for both cases and propose how to unify conflicting types.

4.5.1 Compatible Arrays

The unification in this case may be followed by the recovery of loop structure. Intuitively, the idea is that two or more dimensions recovered from one array reference are treated as only one dimension in a second reference.

```
do i = 1, x
  do j = 1, y
    do k = 1, z
      A(k, j, i) = ...
    end do
  end do
end do
do i = 1, x
  do j = 1, y * z
    A(j, i) = ...
  end do
end do
```

Figure 4.5: Linearized Loop Structure

Consider the example shown in Figure 4.5. We assume that the 2-D and 3-D references to array `A` were both recovered from linearized references and the following mapping vectors were computed:

$$m_1 = \begin{pmatrix} 1 \\ z \\ zy \end{pmatrix}, m_2 = \begin{pmatrix} 1 \\ zy \end{pmatrix}.$$

We see that the references are "compatible" in that the first reference could be changed to use m_2 and the second reference could be changed to use m_1. In the first case, the array `A` would become two-dimensional for both references. In the second case, the unification would make `A` three-dimensional. We can choose either of the solution. Generally, it is desirable to have as many dimensions as possible. However, in some cases, we may find it preferable to choose the lower dimensionality.

For this example if we decide that we need a 2-D structure, we keep m_2 as the mapping vector for `A` and replace the reference in the first loop nest with:

```
A(k + j * z, i) = ...
```

In general with vectors like m_1 and m_2 we will replace the subscript vector $S = \begin{pmatrix} S_1 \\ S_2 \\ S_3 \end{pmatrix}$ with $S = \begin{pmatrix} S_1 + S_2 m_2 \\ S_3 \end{pmatrix}$ and at the same time replace m_1 with m_2.

The alternative unification would have to take care of translating the subscript expression j in the second array reference into a pair of subscript expressions. This is possible without changing the loop structure, but it is better to recover the loop structure at the same time, i.e., to unroll loop j in the second nest into two loops. Code recovery is discussed in Section 5.5.

4.5.2 Unification

```
      L = 0
      IF (NY.GT.1) THEN
        DO 30 K=1,NZTOP
          DO 20 J=1,NY
            DO 10 I=1,NX
              L=L+1
              DCDX(L)=-(UX(L)+UM(K))*DCDX(L)-(VY(L)+VM(K))*DCDY(L)
     1        +Q(L)
   10       CONTINUE
   20     CONTINUE
   30   CONTINUE
      ELSE
        DO 50 K=1,NZTOP
          DO 40 I=1,NX
            L=L+1
            DCDX(L)=-(UX(L)+UM(K))*DCDX(L)+Q(L)
   40     CONTINUE
   50   CONTINUE
      ENDIF
```

Figure 4.6: An example from apsi

Example in Figure 4.6 is extracted from the 141.apsi benchmark from the SPEC CPU95 benchmark suite [63]. Array `UX` (and other arrays in this code fragment) is treated as *either* three-dimensional *or* two-dimensional depending on the value of `NY`.

From loop nest 30, we can recover the following 3-D structure:
`UX(NX, NY, NZTOP)`. The structure recovered from loop nest 50 has two di-

mensions only: `UX(NX, NZTOP)`. It is desirable to have a single type for all references to one array. Because we do not know if `NY` has positive value, the common type for `UX` is: `UX(NX, NY`$^+$`, NZTOP)`, where p^+ is defined to be p if $p > 0$ and 1 otherwise. With this array type we can access `UX` in both loop nests as if it was a 3-D array. The recovered references are: `UX(I,J,K)` for loop nest 30, and `UX(I,1,K)` in loop nest 50.

5 Interprocedural Array Transformations

5.1 Motivation

To change an array layout without changing the meaning of the program we must know if two arrays may access the same memory areas. The natural name for this property would be "aliasing." We have decided to use the term "overlapping" instead since the word "aliasing" has an accepted meaning in programming language research which is not as wide as our meaning of overlapping. We say that two arrays overlap even if those two arrays cannot exist at the same time. Aliasing on the other hand describes a property at a given point of a program execution.

There exist techniques which let the compiler automatically choose[1] an array mapping [17, 4, 36, 42] or distribution [33, 38]. These techniques and our techniques from Chapters 2 and 3 lose much of their effectiveness in the presence of aliasing or more generally in the presence of overlapping. Given an expression which references an array element, a Fortran compiler calculates the address of a memory location which contains that element using fixed, language-specified rules (column-major layout). When the compiler changes the layout of an array, some action must be performed to make references to the same memory area via a different variable consistent with the new mapping. This is necessary to ensure that the transformed program is semantically equivalent to original program. There are two basic approaches to this problem:

- Perform dynamic remapping (redistribution) and ensure that during the time that the layout of an array is different from the standard Fortran layout, the array is not accessed via some other name. To ensure correctness, we still have to perform alias analysis. The required alias analysis is much

[1] The same problem exists of course if the mapping or distribution is specified explicitly by the programmer. Therefore our techniques may be applied with equal effectiveness as an aid for programmers who are willing to explicitly handle array layout.

simpler than the complete overlap analysis tackled by us. This approach has a drawback of extra overhead caused by the dynamic remapping.

- Changes are performed statically, at compile-time. This approach is preferred whenever possible since no extra overhead is incurred. The drawback of this technique that for some programs it is not possible to disambiguate array overlapping.

Hybrid solutions are of course possible. We present an algorithm for performing static changes to array layouts by interprocedural analysis of overlapping arrays. Our algorithm generates an *Array Overlap Graph*. In that graph each vertex corresponds to one array name. There is an edge between two vertices if and only if the two arrays overlap.

The meaning of the graph is that the layout of an array affects all arrays which belong to the same connected component. Note that this is a conservative estimate of actual overlapping. In the following sections we give examples in which two arrays which belong to the same connected component could be remapped independently. We also present techniques which break those "false" paths between vertices of an Array Overlap Graph.

We observe that it is desirable to make the connected components as small as possible. Their size can be minimized by accurate analysis and by selective *procedure cloning*. Our algorithm also coerces the types of all arrays to the same type. We choose an array in a component which has most structure (the largest number of dimensions) and coerce all other arrays to that type. The reason for this is that data remapping and distribution optimizations work best when given most flexibility.

The closest related work to our approach of enabling array transformations by procedure cloning is the work done for using type information to specialize functions in object-oriented languages [56]. However, while the high-level ideas of using type information to enable optimizations by cloning are similar, the issues and resulting algorithms described in [56] are very different from ours.

Rather than starting by presenting a complete algorithm, we will first show a simple case and than gradually refine the algorithm to handle more realistic cases. Section 5.2 shows how to construct an Array Overlap Graph in a simple way. Section 5.3 describes the use of selective procedure cloning to improve the accuracy of the graph. To use the Array Overlap Graph for data layout changes, it is desirable to coerce all overlapping arrays (i.e., all arrays in a connected component) to the type with "most structure." We present an algorithm for type coercion in Section 5.4. Quality of the optimized code can be improved by *code recovery*. This technique is described in Section 5.5. We choose a set of benchmarks whose locality is not optimal and which cannot be optimized by loop transformations due to data dependences and which cannot be optimized by data

remapping without our techniques. We have implemented all those techniques in our compiler based on the Polaris compiler infrastructure [10]. Experimental results which show improvements in speed for those benchmarks are presented in Section 5.6.

5.2 Array Overlap Graph

The main idea of array overlap analysis is similar to existing techniques for detecting *aliases*. The difference is that aliasing considers only variables which are *visible* in the given scope. Here is the formulation of the problem from [25].

> "The problem is to compute, for each variable v and procedure p in the program, a set ALIAS(v, p) of the variables visible in p that may be aliased to v within p."

```
SUBROUTINE CHOTST (ER, FP, TM)
...
CALL COPY (LA, AX, A)
CALL CHOLSKY (IDA, NMAT, M, N, A, NRHS, IDA, B)
...
END

SUBROUTINE CHOLSKY (IDA, NMAT, M, N, A, NRHS, IDB, B)
...
END

SUBROUTINE COPY (N, A, B)
...
END
```

Figure 5.1: Aliasing vs. overlapping.

Consider the example in Figure 5.1. Variables[2] `COPY.B` and `CHOLSKY.A` overlap, but they are not aliased, because they are declared in different scopes.

For straightforward array overlap analysis we could simply use one of the alias analysis algorithms (e.g., the one described in [25]) if we extended it so that arrays in different scopes are being considered. Alias analysis of FORTRAN 77 programs is simpler than general alias analysis due to the absence of pointers.

[2] Our notation prefixes an array name with the procedure name to make clear which array we are referring to. Thus `CHOLSKY.A` means array `A` declared in the subroutine `CHOLSKY`.

In our model we assume that array overlap can be introduced in the following ways:

- Aliasing between a formal parameter and an actual parameter.
- Aliasing between global arrays via the use of common blocks.
- Aliasing introduced by the `EQUIVALENCE` statement of FORTRAN 77.

We represent overlapping arrays with an *array overlap graph*. Each vertex of the graph corresponds to an array (a local variable of a procedure, a formal parameters, or a global variable). An edge is inserted when one of the three situation described above is detected during program analysis.

A connected component in an array overlap graph represents a conservative approximation of all possible names for arrays which are allocated in the same region of memory and share information. Remapping of an array will *possibly* affect other arrays in the connected component. The uncertainty implied by the word "possibly" is caused by the fact that our graph is a conservative, safe approximation of the "overlaps" relation. Each edge corresponds to real overlapping, but the transitivity does not necessarily imply actual overlapping. Let us call the algorithm described in this section the *Simple Algorithm*. Our experiments have shown that the accuracy of the Simple Algorithm is insufficient to perform effective layout changes of arrays in real programs. There are two main reasons for the inaccuracies introduced in the graph by transitivity:

- Different arrays may be passed as parameters to a procedure and will end up in the same connected component via the corresponding formal parameter.
- Arrays may have different sizes. For instance two slices of a bigger array may not overlap directly, but since each of them overlaps with that bigger array, our algorithm assumes that all arrays in the connected component overlap.

The former problem is much more common in real programs, so we describe it more detail and show our solution in Section 5.3. The latter problem can be illustrated by the example shown in Figure 5.2. Arrays `A` and `B` occupy non-overlapping areas in memory. The simple algorithm presented in this section will build a graph with two edges: (`A`, `WORK`) and (`B`, `WORK`) as shown in Figure 5.3. By transitivity, we will conclude that `A` and `B` are overlapping. While for many programs, we have to consider `A` and `B` as "overlapping"[3] even though they do not share memory, in some cases it is possible to use interprocedural array section data-flow analysis to prove that `A` and `B` do not share any information.

[3] They are overlapping in the sense that if we change the layout of `A` then we have to change the layout of `WORK` and consequently we must change the layout of `B`.

```
PROGRAM P
REAL WORK(200)
CALL Q(WORK(1), 10, 10)
CALL R(WORK(101), 100)
STOP
END

SUBROUTINE Q(A, N, M)
REAL A(N,M)
...
END

SUBROUTINE R(B, K)
REAL B(K)
...
END
```

Figure 5.2: Example which shows that the analysis may be inaccurate.

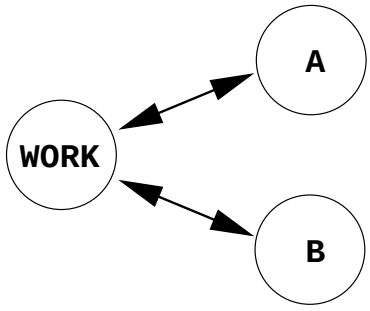

Figure 5.3: Array Overlap Graph

5.3 Improving Accuracy by Procedure Cloning

To make our Array Overlap Graph useful, we have to deal with a very common inaccuracy present in the graphs created by the Simple Algorithm from Section 5.2. In virtually all applications of significant size, it is easy to find a pattern like the one shown in Figure 5.4 (this code fragment, extracted from the NASA7 benchmark, is part of SPEC CPU92 benchmark suite [62]). As can be seen in Figure 5.5, the Array Overlap Graph created by the Simple Algorithm does not allow us to remap the array `CHOTST.B` without remapping `CHOTST.A` at the same time.

Given this code, it is indeed illegal to change the mapping of `CHOTST.B` without other changes to the program. The analysis of the program shows that the best locality of memory accesses could be achieved by changing the layout of `CHOTST.B` and leaving `CHOTST.A` in the column-major mapping (this is due to accesses which are not shown in the included code fragment). We solve this problem by *procedure cloning*. For our working example from Figure 5.4, we create an exact clone of the procedure `COPY` and change one of the calls to use this new version of `COPY`. The transformed code is shown in Figure 5.6. We can see in the corresponding Array Overlap Graph (shown in Figure 5.7) that the dependency between `CHOTST.B` and `CHOTST.A` is gone.

A hasty conclusion from this simple example may be that we should clone all procedures at all call sites. Even a shallow examination of this idea shows

```
      SUBROUTINE CHOTST (ER, FP, TM)
      PARAMETER(IDA=250,NMAT=250,M=4,N=40,NRHS=3)
      COMMON /ARRAYS/ A(0:IDA, -M:0, 0:N),
     $   B(0:NRHS, 0:NMAT, 0:N),
     $   AX(0:IDA, -M:0, 0:N),
     $   BX(0:NRHS, 0:NMAT, 0:N)
      ...
      LA = (IDA+1) * (M+1) * (N+1)
      LB = (NRHS+1) * (NMAT+1) * (N+1)
      ...
      CALL COPY (LA, AX, A)
      CALL COPY (LB, BX, B)
      ...
      END

      SUBROUTINE COPY (N, A, B)
      REAL*8 A(N), B(N)
      DO 100 I = 1, N
        B(I) = A(I)
100   CONTINUE
      RETURN
      END
```

Figure 5.4: Code fragment from CHOLSKY.

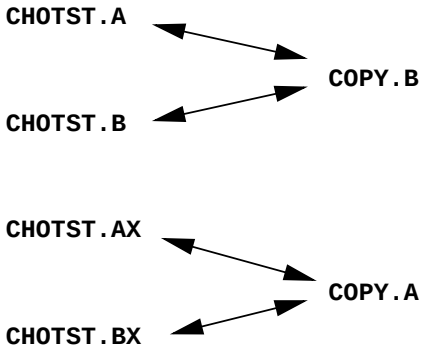

Figure 5.5: Array Overlap Graph before cloning

that not only is complete cloning not possible for languages which allow recursive procedure calls, but it is not desirable to unnecessarily increase the code size of an application even if recursion is not a problem. Larger code size means increased storage requirements and increased load time, but most importantly, it means a larger memory footprint of the application. This of course will likely degrade the

```
SUBROUTINE CHOTST (ER, FP, TM)
...
CALL COPY (LA, AX, A)
CALL COPY1 (LB, BX, B)
...
END

SUBROUTINE COPY (N, A, B)
...
END

SUBROUTINE COPY1 (N, A, B)
...
END
```

Figure 5.6: Code fragment from CHOLSKY after cloning.

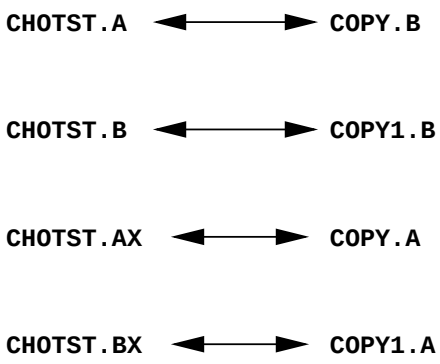

Figure 5.7: Array Overlap Graph after cloning

performance at all levels of memory hierarchies due to increased cache miss ratio and increased virtual memory paging. A small increase in code size is justified if the benefit of changing array layouts outweighs the associated cost.

To alleviate this problem we have developed an algorithm for selective cloning which only clones procedures when doing so will improve the accuracy of the Array Overlap Graph. To further reduce the code size, after the data remapping (or distribution) optimizations have been performed we examine all clones of each procedure and merge those which resulted in the same mapping for all its formal parameters and local variables, and which call equivalent clones at every corresponding call site.

The algorithm uses the concept of an *overlap graph signature* (OGS) which is assigned to every procedure. The basic idea is that the OGS describes the

overlapping relation between formal parameters to the procedure. At every call site we find the overlapping relation between actual parameters and search the list of clones of the callee which have the same relation between its formal parameters. If we find one, we use the clone, otherwise we create a new clone and set its OGS to what we just computed for the actual parameters.

Consider again the example from Figure 5.4. Assume that the following colors have been assigned to the arrays, `A`: 1, `B`: 2, `AX`: 3, `BX`: 4. Color 0 is used for all scalars since they are not affected by array transformations. For every subroutine the compiler maintains a set with all clones created for that subroutine. Initially all those sets are empty—no clones have been created.

During the analysis of the subroutine `CHOTST`, the compiler encounters the first call to `COPY`:

```
CALL COPY (LA, AX, A)
```

The OGS for this call is $(0, 3, 1)$. At this point the set of clones of `COPY` is consulted. That set, $\mathbf{Clones}_{\texttt{COPY}}$, is empty. The new clone is added to the set and the subroutine call is marked to use that clone. The second call to `COPY` follows:

```
CALL COPY (LB, BX, B)
```

The OGS for this call is $(0, 4, 2)$. A clone with that OGS is not present in $\mathbf{Clones}_{\texttt{COPY}} = \{\texttt{COPY}_{(0,3,1)}\}$, so a new clone is created and added to the $\mathbf{Clones}_{\texttt{COPY}}$ set.

After consulting the symbol tables, names for the clones are chosen: `COPY` for the clone $\texttt{COPY}_{(0,3,1)}$ and `COPY1` for the clone $\texttt{COPY}_{(0,4,2)}$ resulting in the code in Figure 5.6 and the Array Overlap Graph in Figure 5.7.

Note that an efficient implementation of the above cloning procedure would apply *lazy cloning* with creating a new copy of the internal representation of a cloned subroutine only if a transformation specific to a given clone is applied. That is the process of "cloning" described in this section would not actually duplicate the representation of a subroutine. Rather a small auxiliary data structure would keep track of clones and their signatures. Our current implementation uses *eager cloning* by actually creating a copy of a subroutine whenever a new OGS is computed at a call site. After the optimizations all the clones are examined and those that were not transformed in any way are replaced with the original subroutine. This process means that the compilation is slower than it could be with a more careful implementation of our algorithm, but the generated code is as efficient as possible with our optimizations.

5.4 Coercing the Types of Overlapping Arrays

So far we have shown how to find a good approximation for the overlapping relationship. We have to do more to change the layout of arrays effectively. This section shows our approach to coercing all arrays in a connected component of the Array Overlap Graph to the same type. Since all overlapping arrays are coerced to the same type, we can guarantee correctness of data transformations by applying the same transformation to all arrays in a connected component of the Array Overlap Graph.

Consider again the code in Figure 5.6. The graph in Figure 5.7 shows that we can safely change the layout of `CHOTST.B` as long as we change `COPY1.B` accordingly. But those arrays have different types: `CHOTST.B` has three dimensions and `COPY1.B`—only one.

For a given connected component of the Array Overlap Graph we find an array with "most structure." In our approach by "most structure" we mean "the largest number of dimensions." Note that it is possible that overlapping arrays have *incompatible* types. Consider as an example `P(3,7)` and `Q(5,4)`—the only common type we could coerce those references to would be a linear (one-dimensional) array.[4] This however would make the array not suitable for layout changes. In that case our analysis gives up and marks all arrays in a connected component with incompatible types as *taboo*. Taboo arrays cannot be subject to any layout changes. This situation is of course possible, but—not surprisingly—very uncommon in real programs.

Coercing array types is performed by propagating types along edges until all arrays have the same type. Note that for aliasing caused by procedure parameter aliasing, the type may be propagated either from the caller to the callee or from the callee to the caller. For other types of aliasing (`EQUIVALENCE` statement and common blocks) the direction of type coercion is irrelevant. The rest of this section shows how the type coercion is performed. We use examples throughout the section to better convey the essence of the algorithm. All examples are taken from standard benchmarks to emphasize the relevance of the discussion to real programs.

We first consider the simpler case in which both arrays start at the same memory address. This case is described in Section 5.4.1. In Section 5.4.2 we extend the algorithm to handle partial array overlapping. Partial overlapping is

[4] We can of course convert any array to any type by first linearizing it and then recovering the necessary structure. For our example, assume that we want to coerce `Q` to have the same type as `P`, so that the new type is now `Q'(3,7)`. A reference `Q(i,j)` would be first linearized to `Q''(i+5*(j-1))` and then the reference using the new type would be `Q'(MOD(i+5*(j-1)-1, 3)+1, (i+5*(j-1)-1)/3+1)`. The index computation would probably become too expensive. However, we expect that this situation is extremely rare.

common due to the practice of passing array slices to procedures which only need a portion of an array.

5.4.1 Arrays with the Same Base Address

We continue with our example from Figure 5.6. First we have to modify the call to `COPY1` to include all values needed to express the type of `CHOTST.B`.

We choose all subscripts of the recovered type to start from the default value for a given language[5] (for Fortran this value is 1). Using a constant lower bound saves us passing the parameters for lower bounds. In this example, the recovered type will be

```
REAL*8 B(NRHS+1, NMAT+1, N+1)
```

In general we would have to add `CHOTST.NRHS`, `CHOTST.NMAT` and `CHOTST.N` to the parameter list, but as an optimization, if array bounds are compile-time constants, we use them directly rather then passing them as parameters. In our example all those values are constant, so the new type of `COPY1.B` is

```
REAL*8 B(4, 251, 41)
```

As another optimization (which is not applicable in this example), we always check if a value is already present in the parameter list before we add a new argument.

We modify all array references by first linearizing the reference and then generating the proper subscripts by using a combination of integer division and modulo operations. In the computation, we use two concepts defined in Section 2.3.1: a *mapping vector* and a *subscript range vector*. Every recovered subscript expression is obtained by first dividing the linearized subscript expression by the mapping vector entry for this dimension and then computing the remainder of the division by the subscript range vector entry for the same dimension (all those operations must be performed on zero-based subscripts). This can in general lead to subscripts which are expensive to compute, but we have developed optimizations which completely eliminate all division and modulo operations for the cases which occur in practice (see Section 5.6 for a partial list of programs used to evaluate our technique).

Two optimizations are used by us to simplify subscript computation:

[5] This is an arbitrary decision with a goal of making the transformed source code more readable for programmers used to the particular language. Any lower bound could be used. All calculations must be made for lower bounds equal to zero, so usually we have to convert the subscript to a zero-based one and then after the necessary calculations, we convert it back to a one-based subscript.

- We eliminate the need to linearize multidimensional arrays before coercing them to the new type. This technique is described in Section 5.4.2.
- We recover the code structure to match the new array type. Section 5.5 demonstrates this technique.

5.4.2 Arrays with Different Base Addresses

Consider the example in Figure 5.8. This code fragment is extracted from the

```
SUBROUTINE DKZMH(DKZM,NX,NY,NZ)
REAL DKZM(NX,NY,NZ)
CALL SMOOTH(DKZM,NX,NY,NZ)
END

SUBROUTINE SMOOTH(F,NX,NY,NZ)
REAL F(1)
MLAG=1
DO II=1,NZ-2
   MLAG=MLAG+NX*NY
   CALL HORSMT(NX,NY,F(MLAG))
END DO
END

SUBROUTINE HORSMT(NX,NY,F)
REAL F(NX,NY)
DO J=1,NY
   DO I=2,NX-1
      F1=F(I,J)
   END DO
END DO
END
```

Figure 5.8: Code fragment from apsi before type coercion.

```
SUBROUTINE DKZMH(DKZM,NX,NY,NZ)
REAL DKZM(NX,NY,NZ)
CALL SMOOTH(DKZM,NX,NY,NZ)
END

SUBROUTINE SMOOTH(F,NX,NY,NZ)
REAL F(NX,NY,NZ)
MLAG=1
DO II=1,NZ-2
   MLAG=MLAG+NX*NY
   CALL HORSMT(NX,NY,F,1+II,NZ)
END DO
END

SUBROUTINE HORSMT(NX,NY,F,SUB,NZ)
REAL F(NX,NY,NZ)
DO J=1,NY
   DO I=2,NX-1
      F1=F(I,J,SUB)
   END DO
END DO
END
```

Figure 5.9: Code fragment from apsi after type coercion.

141.apsi benchmark from the SPEC CPU95 benchmark suite [63]. We want to propagate the type of `DKZMH.DKZM` to `SMOOTH.F` and then to `HORSMT.F`. The first part follows the approach described in Section 5.4.1. We do not have to add new parameters to the call to `SMOOTH` since they are all already present. The reference `F(MLAG)` is converted by Polaris into `F(1+II*NX*NY)` as part of the induction variable elimination. Code structure recovery (see Section 5.5) fails for this loop, so we convert this reference to the new type by using division and modulo. The recovered reference (before simplification) is

```
F(MOD(1+II*NX*NY-1,NX)+1,
  MOD((1+II*NX*NY-1)/NX,NY)+1,
  (1+II*NX*NY-1)/(NX*NY)+1)
```

Symbolic simplification yields `F(1, 1, 1+II)`. Therefore the new modified call statement is:

```
CALL HORSMT(NX,NY,F(1, 1, 1+II))
```

In this example the base addresses of `SMOOTH.F` and `HORSMT.F` are different. We coerce their types by modifying the call to pass the address to the start of `SMOOTH.F` rather than to a slice of it. But now we have to add new parameters to the call to pass the offset into `SMOOTH.F` which is implied by `F(1, 1, 1+II)`. We should pass all subscripts as additional parameters to the call. As an optimization we eliminate compile-time constants and subscripts which are equal to the lower bound in the particular dimension (recall that we always recover types so that they start from 1). In our case the first two subscripts of `F(1, 1, 1+II)` can be eliminated by any of the above optimizations. We only have to add `(1+II)` to the parameter list (this is in addition to passing `NZ` which is needed to declare the array). The new lists of actual and formal parameters are:

```
CALL HORSMT(NX,NY,F,1+II,NZ)
...
SUBROUTINE HORSMT(NX,NY,F,SUB,NZ)
```

In general, to coerce original, two-dimensional array references to the new type *and* account for the lost offset between the base addresses of the two arrays, we have to:

1. Linearize the old reference: `F(1+(I-1)+(J-1)*NX)`.
2. Add the offset: `F(1+(I-1)+(J-1)*NX+SUB*NX*NY)`.
3. Apply the right combination of modulo and division operations and perform symbolic simplification to obtain: `F(I,J,SUB)`. Note that to perform this simplification we have to use the fact that the range of `I` is $(2 \ldots \texttt{NX}-1)$ and that the range of `J` is $(1 \ldots \texttt{NY})$.

The recovered code is shown in Figure 5.9. For this example, it is possible to perform the coercion in a faster way and our implementation uses this as an optimization. Our compiler checks that the dimensions of the old type of `HORSMT.F` are identical to the two leftmost dimensions of its new type. Therefore we can leave the left two subscripts of every reference to `HORSMT.F` unchanged and simply add the missing subscript for the third dimension.

5.5 Code Structure Recovery

This section describes an optimization for a special case which is common enough to deserve a custom treatment.

Consider again the example from Figures 5.4 and 5.6. We have shown in Section 5.4.1 how to coerce the array `COPY1.B` to the type of `CHOTST.B`. If we then use the standard way of transforming every reference to `COPY1.B` by using division and modulo operations, we will obtain the new version of `COPY1` shown in Figure 5.10 (here we assumed that `COPY1.A` was coerced to the type of `CHOTST.BX`). The subscripts to both `COPY1.B` and `COPY1.A` are being calculated in an expensive

```
 SUBROUTINE COPY1(N, A, B)
 INTEGER*4 I, I0, I1, I2, N
 REAL*8 A(4,251,41), B(4,251,41)
 DO I = 1, N
    B(1+MOD(I-1,4),
>    1+MOD((I-1)/4,251),
>    1+(I-1)/1004) =
>  A(1+MOD(I-1,4),
>    1+MOD((I-1)/4,251),
>    1+(I-1)/1004)
 END DO
 RETURN
 END
```

Figure 5.10: Procedure `COPY1` *without* code structure recovery.

```
SUBROUTINE COPY1(N, A, B)
INTEGER*4 I, I0, I1, I2, N
REAL*8 A(4,251,41), B(4,251,41)
DO I0 = 1, 41
   DO I1 = 1, 251
      DO I2 = 1, 4
         B(I2,I1,I0)=A(I2,I1,I0)
      END DO
   END DO
END DO

RETURN
END
```

Figure 5.11: Procedure `COPY1` *with* code structure recovery.

way. We note that trip count of the loop `I` is equal to the size of the arrays. In this special we can replace the loop nest of the depth corresponding to the number of dimensions of the original type (in our example—one) with a loop nest which matches the new type (of depth three in our example). We call this process *code structure recovery*, since it is complementary to the *data structure recovery* from Chapter 4. Procedure `COPY1` after applying code structure recovery is shown in Figure 5.11. This special case is common enough that our compiler recognizes it and performs the necessary code transformation. Currently our implementation attempts it only if the original loop nest is of depth one, the array type before coercion has only one dimension and at least one of the references to that array has a subscript which is equal to the loop index variable.

In our example of `CHOTST.B` and `COPY1.B` in Figure 5.6 our analysis first computes the sizes of both arrays. To do this, we express the size of `COPY1.B` in the terms of variables in the caller (i.e., `CHOTST`). We match the formal parameter `N`

with the actual parameter `LB`. Then we use data-flow analysis to substitute the expression which is used as the actual parameter until it is expressed exclusively in terms of compile-time constants and formal parameters of `CHOTST` (since only compile-time constants and formal parameters may be used in the declaration of array dimensions). If this part of the analysis fails, we are not able to recover the code structure. Once we have expressions for the sizes of `COPY1.B` from before and after coercion, we symbolically compare the two expressions. If their values are equal, we replace the original loop nest with a new one which matches the new type of the array.

Then every array reference which has a subscript equal to the original loop index is replaced by a reference with subscripts which are the loop variables of the new loop nest. At the start of the loop body we also compute the value of the variable which used to be the index variable of the original loop as a function of the new loop variables. This value is used in all other occurrences of the original loop variable. This assignment is not present in Figure 5.6 because it was removed by a dead code elimination phase.

Accessing multidimensional arrays may have higher costs than accessing linear arrays. Therefore in our compiler we record which arrays have been remapped during the optimization phase and all procedures which do not contain references to any remapped array are replaced again with the original procedure. For instance in the Cholesky decomposition kernel, a fragment of which we show in Figure 5.6, initially array `COPY.B` is coerced to the type of `CHOTST.A` and `COPY.A` is coerced by `CHOTST.AX`. However, since none of those arrays is remapped, after the optimization phase, the original version of `COPY` is used again in the call:

```
CALL COPY (LA, AX, A)
```

5.6 Experimental Results

To demonstrate the benefits of our techniques, we found a set of programs which cannot be optimized by existing locality optimizations. Loop transformations cannot be applied because of data dependences and data transformations cannot be used without the techniques described here due to array overlapping. For our benchmarks we had to apply most of the techniques described in this paper. The resulting speedups were obtained by applying unified data and control transformations. We ran the programs sequentially on a DEC AlphaStation 2100 using standard data sizes supplied with the benchmarks and standard optimization flags. The standard data sizes are small for cache sizes found in contemporary computers. For instance, the primary cache of the Alpha processor in our AlphaStation has the size of 16KB. The array sizes in our benchmarks are comparable with the cache size, therefore we expect even bigger performance gains for larger data sets—especially if they exceed L2 cache sizes (1MB on our system).

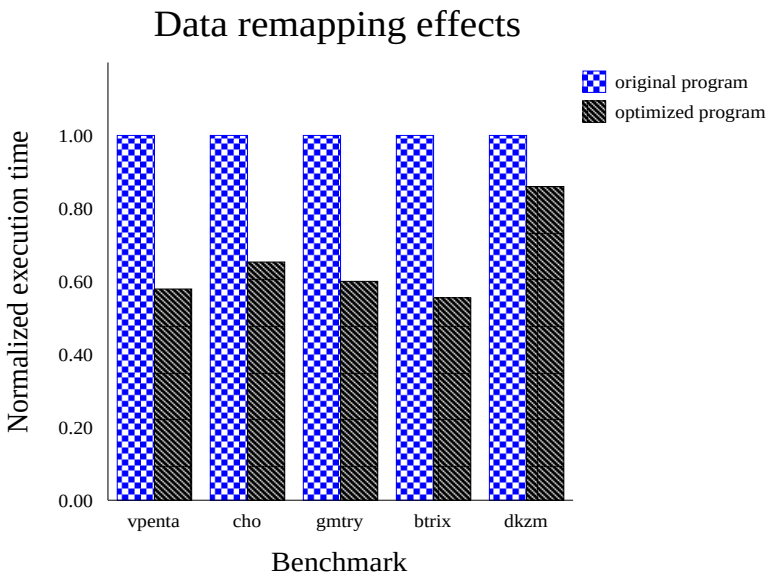

Figure 5.12: Speedups for selected benchmarks

Our benchmarks include four kernels from the SPEC CPU92 benchmark suite [62]: vpenta, cho, gmtry and btrix. The fifth benchmark—dkzm—is a code fragment extracted from the 141.apsi benchmark from the SPEC CPU95 benchmark suite [63]. Our benchmark includes all operations performed on the `DKZM` array. That array is accessed via different names—we had to include five procedures which perform work on `DKZM` and some other procedures which declare that array but never use it in a way other than passing it as a parameter to some other procedure.

Our optimizations cut the execution time by 14%–45%. The relatively small improvement for dkzm can be attributed to the small data size. Standard input data define the array `DKZM` to be of size $64 \times 1 \times 16$ which corresponds to 8KB. The primary data cache in our processor is twice that size, therefore even an unoptimized version of the benchmark ends up having the array in the cache most of the time. As we noted above, we expect even better speedups for larger data sets.

6 Just-In-Time Optimizations for Java

6.1 Motivation

Compiling scientific applications requires sophisticated compiler optimizations. Those optimizations are commonly available in commercial compilers for languages traditionally used for high performance computing. Those languages (the most successful ones are Fortran and, to a smaller extent, C++) are compiled off-line into a stand-alone binary. This approach is different from the dominant compilation model for Java in which the source program is translated into machine-independent `class` files which are then interpreted or dynamically (Just-In-Time) compiled into the target machine language. Mobile code representations which require Just-In-Time (JIT) compilation are becoming very popular as machine-independence and security gain acceptance as desired application properties. Definitions of many contemporary languages include some form of a machine-independent distribution format. The most popular of those languages are: Java [31, 49], Limbo [52] and Omniware [51, 1]. The Tao Operating System [57, 64] has been successfully using a machine-independent model since 1991. An up-to-date list of mobile code projects is maintained by the World Wide Web Consortium [71].

Many of the existing high-performance optimizations cannot be easily adapted to JIT compilation. The reasons include:

- Sophisticated optimization techniques consume a lot of hardware resources: CPU time and memory.
- Often, a global view of the program is necessary (*whole-program optimization* [14, 22]).

In this chapter we address the issue of performing optimizations which are as good or almost as good as traditional optimizations while running much faster and using less memory. We have adapted the technique for Java bytecodes (Section 6.3). However, our experiments with optimizing Java bytecodes described in

Section 6.3 have been performed with an off-line compiler and the compiler algorithms were not as fast as a competitive JIT compiler is expected to be. To perform data transformations we need more program structure than is directly available in the bytecodes. Our off-line compiler was recovering full high-level structure of the original Java program before applying any optimizations. This high-level view of the input bytecodes was represented with an intermediate representation called *JavaIR*. JavaIR was designed by us so that high-performance optimizations, like data remapping, can be performed very efficiently. Unfortunately, our experience has shown that recovering full structure from a `class` file is very expensive. In our implementation, the time needed to convert bytecodes to JavaIR is an order of magnitude longer than the time to perform the optimizations. Since most the high-level structure is not needed to perform data transformation, we have decided to use a different approach in our JIT optimizer.

In Section 6.4 we show how to perform the same optimizations as described in Section 6.3 while recovering only as much structure as needed and using faster (although not necessarily as accurate) analysis techniques than those traditionally used in off-line compilers [2, 70]. We have implemented all the techniques described in this chapter in our experimental compiler *Briki*. Briki uses Kaffe [68] as the JIT compiler and implements the optimizations in a relatively machine-independent manner so that it can run Java programs on multiple architectures (such as i386 or sparc).

6.2 Overview of Array Transformations

Array transformations change the layout of array elements in memory to increase the spatial locality of the application. Spatial locality benefits application by amortizing the cost of fetching a cache line over the uses of array elements co-located in that cache line.

As an example consider an array declared as

```
double A[][] = new double[n][m];
```

The Java language specification [31] and the Java Virtual Machine specification [49] do not define how an array is being laid out in memory. However, the natural implementation of a multi-dimensional Java array would represent `A` as shown in Figure 6.1. This representation is used by Kaffe and many other Java Virtual Machines.

In the loop nest

```
for(int i = 0; i < m; i++) {
  for(int j = 0; j < n; j++) {
```

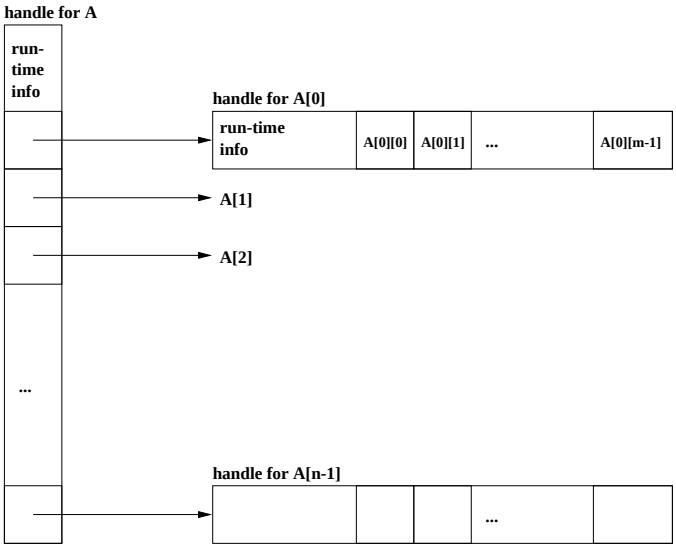

Figure 6.1: Internal representation of a Java array `double[n][m]`

```
    ... A[j][i] ...
  }
}
```

the reference to array `A` exhibits poor spatial locality. References to array elements which occupy consecutive words in memory, e.g., `A[0][0]` and `A[0][1]`, occur in consecutive iterations of the outer loop and are separated by a whole execution of the inner loop which accesses many words in memory and is very likely to replace the cache line fetched in the previous iteration of the outer loop. In our example, even though the reference to `A[0][0]` would bring into the cache the element `A[0][1]`, the reference to `A[0][1]` in the next iteration of the outer loop would suffer the penalty of a cache miss.

There exist two techniques for improving the spatial locality of the reference `A[j][i]` above: loop transformations and array transformations. However the exception mechanism of Java makes performing loop optimizations impossible without sophisticated class hierarchy analysis and interprocedural control flow analysis to ensure that partial results computed in the loop nest are not being used in the exception handlers outside the current method.

Spatial locality of the loop nest discussed in this section can be improved by the array transformation which transposes the two dimensions of `A`. Our optimizations are performed in a JIT compiler without access to the source, but—for clarity—the transformation can be visualized as rewriting the source as:

```
double A[][] = new double[m][n];
...
```

```
for(int i = 0; i < m; i++) {
  for(int j = 0; j < n; j++) {
    ... A[i][j] ...
  }
}
```

The declaration of the array is changed as well as all references to that array. In the optimized code, two consecutive iterations of the inner loop access elements of `A` which occupy consecutive words in memory. For most applications this optimization will result in fewer cache misses and thus a better performance.

Optimization by data remapping must solve two problems:

- Ensuring the legality of an array transformation. Our array transformations permute dimensions of multidimensional array and therefore require the arrays to be rectangular. However, unlike in Fortran or C, a multidimensional array in Java does not need to be rectangular. Our compiler proves that an array is initialized with a rectangular shape and that it is not reshaped during its lifetime.

  The compiler must also make sure that if an array is accessed via different names due to aliasing, all references use the new, transformed type.

- Finding the optimal array layout. It is possible that different references to the same array require different transformations. The compiler must determine what is a globally optimal transformation (we do not allow dynamic reshaping of arrays).

Both issues are addressed in the next section.

6.3 Off-line Optimizations

6.3.1 Intermediate representation

The intermediate representation is a syntax tree with a node for every Java class defined in the input. Every class node contains references to nodes for all interfaces, fields and methods defined for that class as well as some other information (access flags, a reference to the super class etc.). *JavaIR* is our implementation of this intermediate representation. An overview of JavaIR is presented in this section. More details on JavaIR can be found in [18]. Section 6.3.2 discusses some issues related to the problem of the recovery of the structure needed to build JavaIR from the bytecode representation of a program.

In the recovery process we attempt to achieve a structure which is as close to the Java source as possible. For the benchmarks we have used, our IR corresponds directly to a Java source form of the same benchmark. It is however possible that a `class` representation of an application either does not correspond to a legal Java source or it would be impractical to recover the source form. Section 6.3.2 is devoted to those problems.

Figure 6.2 shows a class hierarchy for a subset of the classes used in JavaIR.

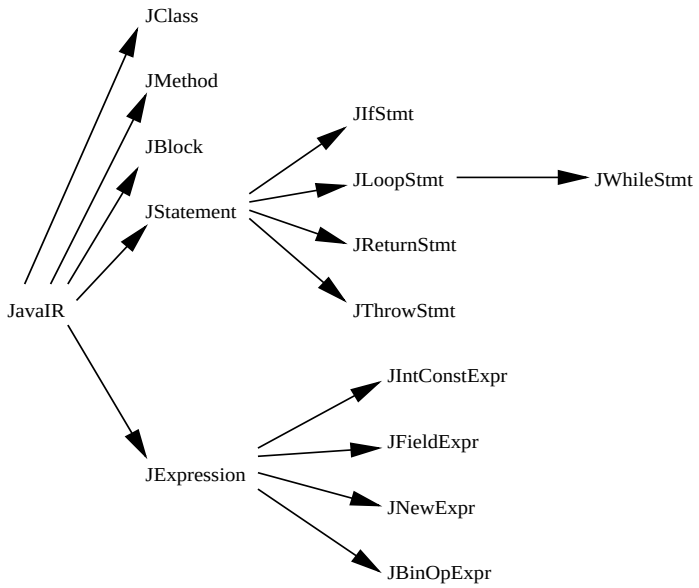

Figure 6.2: Class hierarchy of the nodes in the JavaIR intermediate representation (only selected expression and statement types are shown)

We will use the following, very simple example to illustrate how JavaIR represents Java programs.

```
class simple {
   int i;
   public simple() {
      i = 0;
   }
} //simple
```

A complete JavaIR representation of class `simple` takes care of many details that are required to maintain the meaning of the program. The image gets even more complicated by the fact that some information is replicated so that it can be accessed quickly.

The JavaIR representation is not *exactly* the same as the original source. Instead, JavaIR corresponds closely to the code generated by the compiler. In our

example, the code for the constructor contains *implicitly* two additional statements:

```
public simple() {
  super();    // implicit call
  i = 0;
  return;     // implicit statement
}
```

All implicit methods, statements and expressions are represented explicitly in JavaIR.

All nodes in the syntax tree are objects of some class derived from the class `JavaIR` (see Figure 6.2). The intermediate representation defines classes which are not derived from `JavaIR`, but none of them can be used as a node in the tree. A Java class is represented with an object of the class `JClass` which contains lists of objects representing fields, methods and interfaces. A method is represented with an object of the class `JMethod`. If the method is not native, this object contains a reference to an object of type `JBlock` which in turn contains a reference to the symbol table and a list of statements. A statement may contain other statements or expressions. Fields and interfaces follow the similar idea, but their representation is much simpler.

6.3.2 Recovering high-level structure

One of the important features of our compiler is the ability to recover high-level structure of a source Java program given its bytecode representation. In this section we discuss the process of that recovery as performed by our bytecode to JavaIR front-end. We start by giving a simple example of a bytecode sequence and the recovered source code. Note that our front-end actually recovers an internal representation which is not directly printable. However, since for our applications there is a straightforward mapping between the JavaIR described in Section 6.3.1 and a Java source, we simply show the recovered source code. The rest of this section shows a selection of more interesting problems which must be addressed for the recovery to take place. The problem of high-level structure recovery is very similar to the problem of decompilation. Previous work on decompilation addressed this issue primarily for reverse-engineering reasons. While our goal is different (we want to recover the high-level structure to enable some compiler optimizations), we borrow from existing decompilation techniques in our design. Existence of many decompilation tools shows that high-level structure recovery from low level code is possible. Indeed there exists a decompiler for Java [65] which given a `class` file will produce its representation as Java source. Another interesting decompiler is described in [24]. In our work, we have identified the

following problems. They are all discussed in more detail in the remainder of this section.

- Converting a stack-based code without branches to a syntax tree. This is generally straightforward. The only problem that required extra attention was the use of temporary variables during the evaluation of a complex expression on the stack. Dataflow analysis is needed to ensure that flow dependences are not violated.

- Identifying types of local variables. Again, dataflow analysis is used to disambiguate types of variables. The disambiguation may be needed because either the same variable may be used to store values of different types or assignments which are allowed in the bytecode must be changed to narrow down expression types (e.g., Boolean values are represented internally as integers, but in JavaIR, we want to represent them as Boolean values).

- Converting short-circuit operators into expression form. This is performed by a flow patterns recognition mechanism similar to the one used in [24].

- Converting branches to high-level statements (loops, conditional statements, `break`, and `continue` statements). We have based our approach on the algorithm presented in [5]. Some modifications to the algorithm were necessary to more completely eliminate branches.

Our general approach is to first recover simple expressions and statements within basic blocks (bytecode sequences without branches). This process creates a *control flow graph* (CFG) with high-level constructs in its nodes. Then we convert CFG edges to structured control flow constructs using techniques described later in this section.

An example

Consider the following code fragment of a disassembled `class` file. It is not apparent what this sequence of bytecodes does.

```
 0 iload_1
 1 iconst_1
 2 if_icmpeq 10
 5 iload_1
 6 iconst_3
 7 if_icmpne 18
10 aload_0
11 iconst_1
```

```
12 putfield #4 <Field cond1.i I>
15 goto 23
18 aload_0
19 iconst_2
20 putfield #4 <Field cond1.i I>
23 aload_0
24 dup
25 getfield #4 <Field cond1.i I>
28 iconst_1
29 iadd
30 putfield #4 <Field cond1.i I>
```

One can of course understand this code with some effort. Most traditional compiler optimizations can operate on an intermediate representation which closely corresponds to the bytecode.

Our compiler algorithms work best when presented with high-level constructs like a for-loop or a multi-dimensional array reference. Neither of those two constructs is explicit in the bytecode form, which implements them with low level operations. In many cases it is however possible to recover a representation of a method with those high-level constructs.

Consider again the above sequence of Java VM instructions. Our compiler parses it and generates an equivalent JavaIR representation which can be printed as the following Java source fragment (as described earlier the internal JavaIR representation is not directly printable since it is a collection of Java objects with references to each other).

```
if(((a1 == 1) || (a1 == 3))) {
   this.i = 1;
} else {
   this.i = 2;
} //if
this.i = (this.i + 1);
```

Note that `a1` is used to represent the first argument to the method represented by this sequence of bytecodes. The identifier used by the programmer in the source cannot be recovered.

Extracting expressions and simple statements

The design of Java bytecodes makes it relatively easy to recover expressions and simple statements (assignments and method invocations). Recovering statements which change the control flow requires more work since their implementation in

the bytecodes uses branches which we do not want to use in JavaIR. We show in later how to convert branches to structured control flow constructs.

To convert a sequence of bytecodes contained in a basic block to high-level expressions and statements, we symbolically execute each basic block using a temporary stack to emulate a Java virtual machine. For instance the following basic block from the example shown earlier in this section

```
23 aload_0                          ; push ``this'' on the stack
24 dup                              ; duplicate the top of stack
25 getfield #4 <Field cond1.i I>    ; push field i on the stack
28 iconst_1                         ; push 1 on the stack
29 iadd                             ; add the two top values
30 putfield #4 <Field cond1.i I>    ; move top of stack to field i
```

is converted to

```
this.i = (this.i + 1);
```

To see how the symbolic emulation works, consider the instruction `iadd` above. Processing of the previous instructions have placed two references at the top of our symbolic stack, one for the expression representing the constant 1 and another one for the expression `this.i`. Our front-end creates a new JavaIR object for integer addition and initializes its two operand fields to the two values popped from the top of the stack. The new object is being pushed on the top of the stack.

This approach generally works for our benchmarks. The only interesting problem that must be solved is caused by using the stack for storage of variable and field values which have been overwritten during a calculation of a complex stack expression. Consider the example in Figure 6.3. The sequence marked with `x` is converted to

```
this.i = 1;
```

and the sequence marked with `z` is converted to

```
this.j = (this.i + this.i);
```

but the first occurrence of `this.i` on the right-hand side of the second assignment should have the value that the field `i` had *before* the assignment of 1, and the second occurrence should have the value that the field `i` had *after* the assignment of 1. Our solution keeps track of the flow of values and when necessary creates temporary variables to hold the values that the Java VM stores on the stack. For the example from Figure 6.3, we recover a sequence corresponding to the following source fragment:

```
aload_0                              z
getfield #4 <Field cond1.i I>        z
aload_0                           x
iconst_1                          x
putfield #4 <Field cond1.i I>     x
aload_0                              z
getfield #4 <Field cond1.i I>        z
iadd                                 z
aload_0                              z
putfield #5 <Field cond1.j I>        z
```

Figure 6.3: A store during expression evaluation

```
tmp = this.i;
this.i = 1;
this.j = (tmp + this.i);
```

Short-circuit operators

Consider again the bytecode sequence shown on page 77. An alternative representation called a control flow graph (CFG) is shown in Figure 6.4. A CFG contains

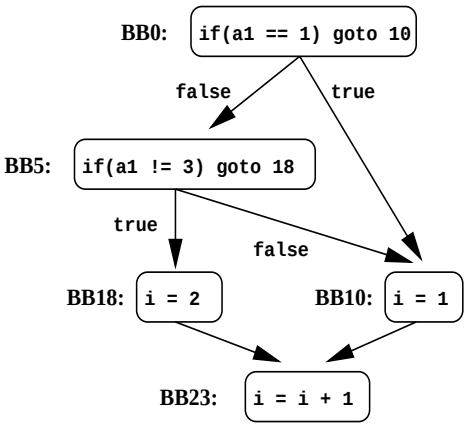

Figure 6.4: A control flow graph for a short-circuit operator

a node for every straight-line code sequence (a *basic block*) and an edge for every branch (a conditional branch results in two edges, an unconditional branch corresponds to a single edge). Nodes in Figure 6.4 are labeled with the numbers corresponding to the address of the first instruction of every basic block. Labels on the edges mark "true" and "false" branches.

Our recovery mechanism recognizes patterns in the CFG corresponding to different short-circuit operators in the manner similar to the technique described in [24].

Goto elimination

This step is based on the algorithm described in [5]. We have extended this algorithm to eliminate branches in some cases in which the original algorithm failed. Nevertheless, for some CFG's we are not able to completely eliminate branches. For reducible flow graphs, branch elimination is always possible, but sometimes it requires inserting new loop statements or duplicating code. While those approaches would be acceptable if our goal was to recover legal Java source for the widest possible range of `class` files, adding new loops or duplicating code would not help us in any high-level compiler optimizations. Therefore, we decide to allow branches in JavaIR at the cost of not being able to optimize those programs with the current, preliminary version of Java (since our compilation process involves compiling Java source generated from JavaIR). Branches are not a problem in the JIT version of Briki which is described in the next section.

Our implementation of branch elimination is implemented with the following algorithm:

1. Collapse short-circuit operators.
2. Find natural loops.
3. Insert repeat nodes.
4. Find dominators, heads and follow sets.
5. Expand loops.
6. Call `getform`.
7. Call `addbranch`.
8. Clean the graph (push loop termination conditions into loop headers, unify types, etc.)

Difficulties in structure recovery

The implementation of Briki described in this section parses a `class` file, represents it in JavaIR, performs the desired optimizations and prints the optimized source to a file. We later use a standard compiler to convert the resulting source file to the bytecode form. The off-line process was chosen by us for the ease of

debugging and more flexibility experimenting with new optimization techniques. The JIT version of Briki described in Section 6.4 performs the same optimizations on a simplified IR, and then directly generates machine code from the optimized IR rather than generating the source first.

For the `class` files we examined, it is possible to efficiently recover a JavaIR form of those applications which can be printed out as legal Java source. However, there exist examples of `class` files for which this process would be expensive or would not yield any benefit for our optimizations techniques. It is important to stress here that for our optimizations we are interested in recovering a *high-level structure*. Some bytecode sequences do not have such structure and although it would be possible to for instance eliminate all branches and generate legal Java source for such programs, the *structure* created by the branch-elimination process would be artificial and would not enable any additional high-level optimizations.

We also note that it is possible that a given application distributed in the `class` file form would not be representable as legal Java source (one such example is given in Section 13.4.6 of the Java language specification [31]). While this would cause problems for the current implementation of Briki, it would not affect the JIT version of our compiler.

Recovery algorithm

We sketch here the process of high-level structure recovery. We assume that all data structures for classes and class members have been initialized.

The algorithm performs the following steps for every non-native method.

1. Scan all opcodes and mark basic blocks.

2. Parse the exception table (see Section 4.7.4 of [49]) to mark portions of the bytecode stream handled by specific handlers and to mark the handlers themselves.

3. Create a control flow graph (CFG) for the method. The CFG has at least one *entry node*—for the main entry to the method—but may also contain additional entry nodes for every exception handler of that method.

4. Structure the CFG by identifying short-circuit operators and creating high-level control flow nodes as described in Sections 6.3.2 and 6.3.2 accordingly.

5. Starting from the entry nodes, visit all nodes of the CFG and interpret the code of each of them. This step creates the JavaIR nodes for every basic block. Every node of the structured CFG has two lists of JavaIR nodes: statements and expressions. The expressions are needed for values passed

on the Java VM stack across basic block boundaries. The values left on the stack at the end of a basic block are being used in the successor nodes as initial stack content.

6. The information extracted in the previous two steps is used to create the complete JavaIR form for this method.

6.4 Just-In-Time Optmizations

Performing high-level optimizations in a Java JIT compiler is more difficult than in a traditional compiler. The main reasons are:

- Much of the information about program structure is lost in the process of translation from Java source to bytecodes. This information (e.g., multidimensional array references) must be recovered before any optimizations can be attempted. In our particular implementation that recovery is even more difficult since, for performance reasons, we have decided to directly use the Kaffe IR which is at lower abstraction level than bytecodes.

- Since the compiler is being invoked every time the program is being run, the speed of compilation is much more critical and many traditional analysis techniques must be replaced with faster alternatives. Such new algorithms are proposed in this section.

- Typically, to improve response time, a JIT compiler translates bytecodes on demand one class at a time or one method at a time (Kaffe uses the latter approach). Therefore, optimizations which require global information are not possible (or at least cannot be trivially implemented).

Since our compiler is embedded in Kaffe we have decided to use Kaffe IR directly. A brief overview of the Kaffe architecture is presented in Section 6.4.1. Briki performs the optimizations in the following steps:

1. Build a control flow graph (CFG) for the method.

2. Find dominators.

3. Identify loops.

4. Compute def-use information.

5. Compute *constant values*.

6. Find *loop-defined variables*.

7. Identify multidimensional arrays which can be legally remapped.
8. Find optimal mappings.
9. Remap arrays.

The rest of this section will discuss our implementation decisions for the above algorithm.

6.4.1 Kaffe architecture

Kaffe [68] is a free JIT implementation of the Java Virtual Machine [1]. Kaffe runs on different architectures: i386, Sparc, Alpha, M68K. This is possible because Kaffe is designed to have two parts:

- the machine-independent front-end which translates bytecodes into a low-level intermediate representation, *Kaffe IR*, and
- a machine-dependent back-end which translates Kaffe IR into the required machine language.

Kaffe IR is designed to closely resemble a modern microprocessor architecture so that writing a new back-end is relatively simple. Kaffe IR defines a virtual architecture with a very large number of pseudoregisters (every local variable and every stack location is mapped to a different pseudoregister) and it contains instruction to move data between registers[2] or between registers and memory, perform arithmetic and logical operations on values in registers, branch conditionally to a label, call a subroutine, etc.

Every back-end must provide a set of core virtual machine instructions. If a virtual instruction cannot be mapped directly to the target instruction set, it is implemented in software. A back-end may also implement one of optional instructions which will be used by Kaffe to implement some bytecodes more efficiently.

A Kaffe IR (virtual) instruction is represented internally as a `sequence` structure with a pointer to the back-end function which takes as a parameter a pointer to the `sequence` and translates it into the target instruction set. The `sequence` structure also contains all operands (pseudoregisters, constants or labels) to that virtual instruction.

The translation process consists of two basic steps. First the bytecodes are translated by the front-end into the Kaffe IR, then the list of the `sequence` structures is traversed and every back-end function is called generating native code.

[1] Kaffe can also run as an interpreter, but we use it only in its JIT mode.

[2] We will use the terms *register*, *pseudoregister* and *variable* interchangeably.

The developers of Kaffe are very good about releasing updates to Kaffe. In the 14 months since the Kaffe project started, there have been 21 releases of Kaffe (the latest one is 0.8.3), or—on average—one every 3 weeks. This caused a technical problem for us, since the most convenient and efficient way of implementing our optimizations would involve modifying internal data structures of Kaffe, but to keep in sync with the updates of Kaffe we would have to merge our software with the new Kaffe every three weeks, thus stretching our resources. To avoid delays, we have decided to keep our optimizer as separate from Kaffe as possible. While this made our project manageable, it resulted in extra overheads both in memory and processing time. In that light we are glad that our optimizations are very fast (see Section 6.5 for timings), because there is room for additional improvement which can be achieved by a tighter integration with Kaffe. Main sources of those, unnecessary, overheads are

- Need to allocate a shadow data structure for every `sequence` structure to keep additional information needed during optimization. Splitting the information into two structures results in lost spatial locality and unnecessary indirection.
- Need to recognize individual instructions in Kaffe IR. Since the only way to tell what instruction is implemented by a given `sequence` structure is to compare the back-end function pointer against the addresses of back-end functions extra time is spent where a single instruction would suffice if we could change Kaffe IR to suit our needs.

In the current implementation, we let Kaffe front-end generate its IR (the `sequence` structures), then we invoke our optimizer which analyzes and transforms Kaffe IR, and after the optimization is done, we return to Kaffe and the back-end translates the optimized Kaffe IR form into native code. That design means that the only change to Kaffe source code needed to plug in our optimizer is the insertion of a single function call just before the back-end invocation.

6.4.2 Control Flow Graph

We build the CFG for the Kaffe IR form of a method by searching for the `startBlock` and `endBlock` instructions. Than we use standard algorithms to find dominators and identify loops [2]. Those algorithms take about 10% of the total optimization cost (compare Figure 6.7).

6.4.3 Computing Def-Use Information

This step computes def-use information for every basic block. The operation is very simple and requires visiting every Kaffe IR instruction exactly once to per-

form a sequence of simple bit operations. In our implementation this is the most expensive part of the array transformation optimization (compare Figure 6.7).

The long time to perform this optimization is an artifact of our software engineering decision (the independence of our code from the changes in Kaffe sources) explained in Section 6.4.1. For relative independence of Briki from changes in Kaffe we pay the price of

- The need for keeping a shadow structure for every Kaffe IR instruction (which results in worse cache utilization and unnecessary indirection in accessing the structures).
- Very slow code to determine which registers are being modified in a given instruction.

We think that the performance of Briki embedded in Kaffe can be improved when Kaffe becomes more stable and a tighter integration of Kaffe and Briki becomes possible.

6.4.4 Computing Constant Values

For every array reference we have to be able to identify which array allocation statement is associated with this reference. There are many known algorithms for solving this problem of *reaching definitions*. Existing algorithms are not well suited for JIT compilation since they use large amounts of memory to store the reaching definitions information (e.g., in the form of bitmaps for every instruction or *ud-chains* [2]).

For our purposes we solve a simpler problem which can be computed faster and represented in a more compact way. The representation contains one bit vector per basic block. The bit vector represents *constant values* with the following definition. A variable v is constant-value for a basic block B if and only if

1. v is defined in B.
2. v is not defined in any other basic block.
3. v is used only in basic blocks *dominated* by B.
4. v is not used before its definition in B.

We only analyze and possibly transform arrays which are stored in constant-value variables. Elements of the array may of course be modified many times, but we make sure that the variable containing the pointer to the array handle is

a constant-value variable. Note that the fact that an array is stored in constant-value variable does not suffice to determine if an array transformation is legal. For example a shape of a 2-D array could be changed by changing the length of one of the rows, or a row could be passed as an argument or returned as a result. We conservatively assume that only arrays which have been allocated as rectangular arrays (using the construct corresponding to the `multianewarray` bytecode) and whose all uses are *references using all dimensions* can be remapped. This condition ensures that arrays optimized by us remain rectangular during their lifetimes.

We can find constant-value arrays efficiently. For every array allocation, we note the variable, v, the array is stored in, and the basic block B that contains this allocation and we traverse all other basic blocks and make sure that

- v is not defined.
- v is used only in basic blocks which are dominated by B.

The above operation can be performed as a single scan of all basic blocks using the def-use information computed for every basic block in Section 6.4.3. During the same scan we verify that all uses of v access the array using all its dimensions.

Note that for block B we have to traverse all its instructions rather than just use the summary def-use information computed for B in Section 6.4.3.

6.4.5 Identifying Loop-defined Variables

To analyze precisely locality properties of array references contained in loops, we would have to determine which variables are loop induction variables and what is their step for the corresponding loops. Again, this operation is expensive and we have decided to approximate the notion of an induction variable with a *loop-defined variable*. For a loop L, a loop-defined variable is any variable which is defined in loop L. This simplification is justified by an observation that for scientific programs if a variable is assigned a new value in a loop then the value of that variable will be different in every loop iteration and the variable will usually be an induction variable with a unitary step.

Loop-defined variables can be identified in a single scan of all basic blocks. For every basic block the set of variables defined in this block is added to the set of loop-defined variables for all (if any) enclosing loops. Enclosing loops can be accessed quickly because every basic block has a pointer to the innermost loop containing this block and every loop has a pointer an enclosing loop.

6.4.6 Recovering Multidimensional Array Structure

Array transformations can be only applied to multidimensional arrays. Multidimensional array references are present in the source program, but the translation to the bytecode form lowers them into a sequence of one-dimensional references. Those can be converted back into multidimensional references with the techniques described in Section 6.3.

The dimension recovery is more difficult in the JIT version of Briki. While in principle we could operate on bytecodes, for reasons explained in Section 6.4.1, we perform our optimizations on Kaffe IR which is even lower than bytecodes. In Kaffe IR, an array reference is converted to a sequence of low-level instructions like memory loads, register moves, shifts and additions. Briki analyzes each such sequence and represents it as a high-level multidimensional array reference. This high-level structure is not stored anywhere—it is recomputed every time it is needed. The impact of recomputation is negligible and our optimizations are very fast. For cholesky, the time to ensure legality of a transformation, finding the optimal mapping (Section 6.4.7) and performing the transformation (Section 6.4.8) constitutes only about 20% of the time spent in Briki.

6.4.7 Finding Optimal Mappings

Since multidimensional arrays in the Kaffe VM are not represented as contiguous portions of memory, but rather as arrays of 1-D arrays, we use a much restricted set of array transformations as compared to what would be possible with the framework from Chapters 2 and 3.

For every array reference we determine which subscript contains loop-defined variables of the innermost loop (or of the next enclosing loop, if the loop-defined variables of the innermost loop are not present in any subscript).

The best mapping for this array would make the dimension which corresponds to this subscript the right-most dimension to increase spatial locality.

In any real program it is very likely that there would be conflicts between desired mappings for different array references. We resolve the conflicts by assigning priority to every array reference. The priority is based on the loop nesting—the higher the nesting the higher priority. This policy is adopted since *usually* the statements with higher nesting are executed more times than statements with lower nesting. This is of course not guaranteed and in general it is not possible to predict at compile-time how many times a given statement will be executed.

In the case of a conflict we choose the array mapping preferred by the greatest number of references with the highest priority. For instance, if four is the highest priority and three references with priority four require dimension 0 in the right-

most position and one reference with the same priority requires dimension 1 in the right-most position then dimension 0 is permuted into the right-most position.

6.4.8 Remapping Arrays

Once we know which dimension should be permuted to the rightmost position, remapping is very simple. In this process we use the dataflow information calculated in the step described in Section 6.4.3.

Before we apply our transformations, we have to ensure that they do not change the meaning of the program. There are three issues we have to deal with for our transformations.

- Arrays we remap must be rectangular.
- The order of evaluation of a remapped array reference must be the same as for the original expression (see Section 15.12.1 of the Java language specification [31]).

The rest of this section gives more details about each of those three potential problems.

Array shapes

Array remappings used in our data and code transformations framework [17] can be applied only to multidimensional array which are *rectangular*. In a rectangular array all elements in a given dimension have the same size. Compiler-supported arrays in languages like C, C++ or Fortran are rectangular (of course a programmer may implement non-rectangular arrays explicitly by linearizing the logical structure in a one-dimensional array). Multidimensional arrays in Java are more general since they are treated as arrays of arrays.

For our optimizations, we check the array was allocated using the `multinewarray` opcode and that all accesses to that array use all dimensions. This approach is more conservative than necessary, but it is very fast and completely adequate for scientific codes.

This condition is sufficient because to change the shape of an array or to use an array in a way that depends on the mapping, a reference to one of the subarrays would have to be used. E.g., for an 3-D array `B`, if all references *use all dimensions*, i.e., are of the form `B[`*expr1*`][`*expr2*`][`*expr3*`]`, the mapping of the array cannot change the semantics of the optimized application. However, expressions of one of the following three forms, `B[`*expr1*`][`*expr2*`]`, `B[`*expr1*`]`, or `B` could potentially depend on the mapping and arrays with expression that *do not use all dimensions* are not transformed by Briki.

Array expression evaluation order

The definition of Java specifies that a multidimensional array reference must be evaluated from left to right. Our simplest remapping switches two dimensions thus changing the evaluation order.

We solve this problem by checking if subscript expressions have potential side effects (note that a possibility of an abrupt completion is a potential side effect). Those subscript expressions that may have potential side effects, are being evaluated in the original order with the results being stored in local variables rather than being stored on the stack directly.

6.5 Experiments

We perform our optimizations using the kaffe JIT compiler [68]. We run all experiments under Linux 2.0.25 on a computer equipped with a 200MHz Pentium Pro processor.

6.5.1 Array layout optimizations

We test our optimizations on programs ported by us directly from their Fortran versions included in the SPEC CPU92 benchmark suite [62]: mxm, vpenta and btrix.

Memory locality optimizations rely on high relative cost of memory references. In inefficient code, the effect of improvements in locality is negligible due to long computation time. Our optimizations are very effective for languages with good compilation technology. For interpreted Java programs the effect is not noticeable as evidenced in the table showing running times in milliseconds:

appli-cation	problem size	interpreter (JDK)		JIT (kaffe)	
		unopt	opt	unopt	opt
mxm	200	35758	35687	2936	2624
cholsky	250	49817	49450	11926	9974

Even for JIT Java virtual machines, the improvements for array-based optimizations are significantly smaller than improvements for the same programs implemented in programming languages like Fortran and C. We attribute that to relatively immature compiler technology for Java. Even a simple access (indexed by loop variables, like `a[i][j]`) to an element of a two-dimensional array is translated by Kaffe to a sequence of many instructions including two conditional branches and 11 memory operations (!) as shown in Figure 6.5.

```
movl 0xc(ebp),edx
movl edx,ebx
movl -0x24(ebp),eax
movl eax,ecx
movl ebx,edx
addl $0x8,edx
movl 0x0(edx),edx
cmpl edx,ecx
jl   skip1
call 0x40015e40
skip1:
movl ecx,edx
shll $0x2,edx
addl $0x20,edx
addl ebx,edx
movl 0x0(edx),ebx
movl -0x20(ebp),eax
movl eax,ecx
movl ebx,edx
addl $0x8,edx
movl 0x0(edx),edx
cmpl edx,ecx
jl   0x80b2d6f
call 0x40015e40
skip2:
movl ecx,edx
shll $0x3,edx
addl $0x20,edx
addl ebx,edx
fstpl -0x14(ebp)
movl ecx,-0x1c(ebp)
movl ebx,-0x18(ebp)
fldl 0x0(edx)
fstpl -0x1c(ebp)
fldl -0x14(ebp)
faddl -0x1c(ebp)
fstpl -0x14(ebp)
fldl -0x14(ebp)
addl $0x1,eax
fstpl -0x30(ebp)
movl eax,-0x20(ebp)
```

Figure 6.5: Code generated for `a[i][j]` by Kaffe for the i86 architecture

The same loop written in C and compiled with "`gcc -O3`" results in a single instruction:

```
faddl a(eax)
```

We present normalized running times for our set of benchmarks in Figure 6.6.

For our array-based benchmarks, the JIT version of briki performs the same optimizations as its off-line predecessor. Furthermore for our data sets the time to perform the optimizations just-in-time is several orders of magnitude shorter than the time to run the benchmark, even though we chose small (by scientific computing standards) problem sizes—none of our benchmarks took more than 15s to complete. Since the overhead for JIT optimizations is so small, the speedups are identical (within three significant digits) with the results obtained with the off-line version of our compiler [21] and range from 10% to 20%.

The speedups of Java benchmarks are not as good as the results of applying the same optimizations to Fortran versions of the same programs (those were as high as 50%). This can be explained by the lack of standard optimizations in the pre-release version of Kaffe used in our experiments. The quality of code generated by Kaffe will, undoubtfully, improve over time. The current version

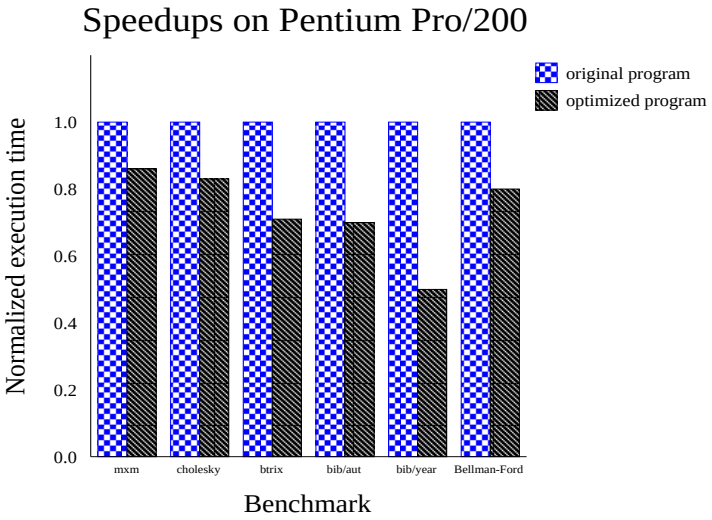

Figure 6.6: Speedups achieved on a Pentium Pro/200 PC

produces very poor code for array references. For example the i386 instruction sequence for a simple reference to a 2-D array: `a[i][j]` contains 30 instructions including two branches (for bounds checking) and eleven memory references (for accesses to array handles and register spills and reloads). A high-performance Fortran compiler would translate an array reference like that (if `i` and `j` are loop variables) to one or two instructions with just one memory reference—the load of the array element. Our optimizations reduce the cost of this one load, by increasing the cache hit ratio for this instruction. In the code generated by Kaffe this gain is dampened by the cost of the extra nine memory references and 20 other instructions which are not improved by our optimizations.

To provide a better context for the discussion in Section 6.4, we present here times spent in various steps of our optimizer. Some of the times measured by us were too short for the granularity of standard Unix timing routines. To obtain accurate results, we have used cycle counters defined by the Pentium architecture. The resolution of those counters was more than sufficient for our needs.

Figure 6.7 shows the break-down of the array transformation time for our cholesky benchmark. The time represented in the figure corresponds to less than half of the total JIT time (if we include the time of the Kaffe front- and back-ends). The absolute time for the array transformation optimization on a 200 MHz Pentium Pro is 8.4 ms. Figure 6.8 shows the overhead for performing array transformations for Java versions of cholesky and mxm kernels from the SPEC CPU92 benchmarks [62]. Note that we have increased data sizes so that the execution times are meaningful. For scientific computing standards the problem sizes are still very small: the execution times are 4 s for mxm and 12 s for cholesky. However, the array sizes are large enough to benefit from better locality.

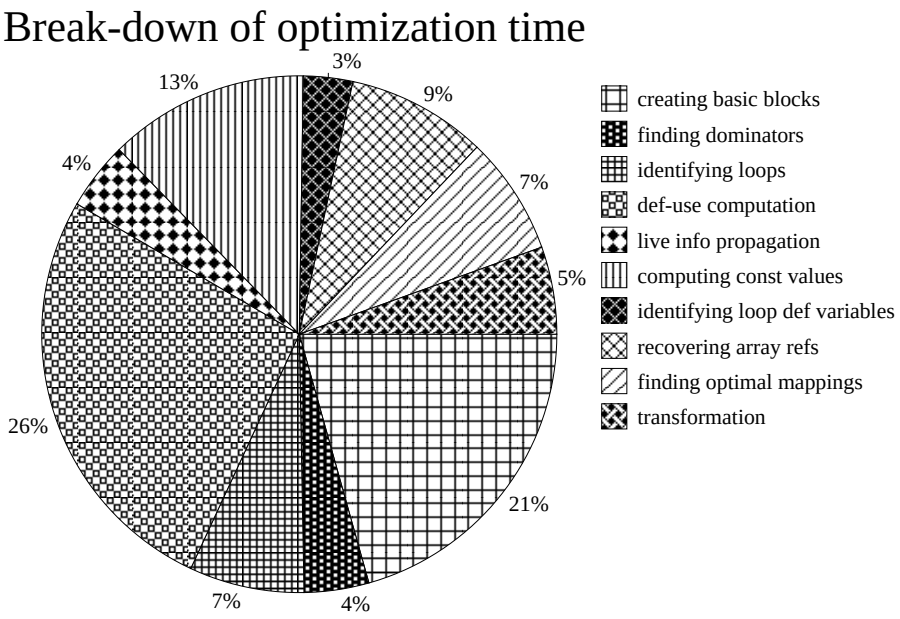

Figure 6.7: Break-down of the optimization time for cholesky

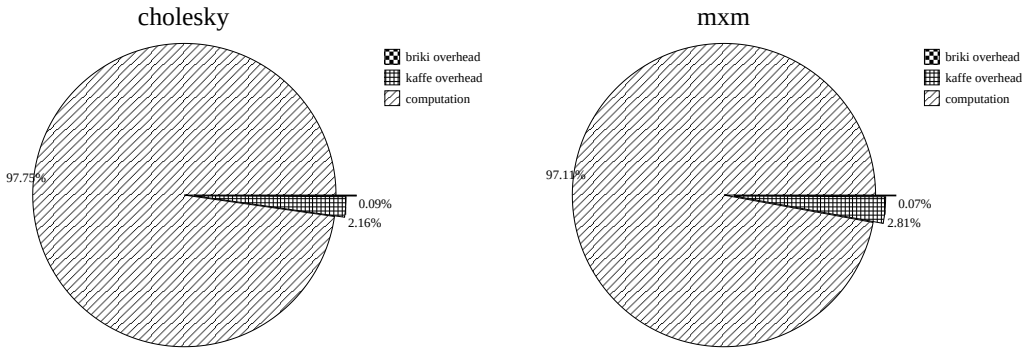

Figure 6.8: Briki vs. kaffe

Even for those very small problem sizes, the relative time spent on our optimizations is completely negligible—three orders of magnitude shorter than the time spent on useful computation. Note that the time spent in Kaffe *translating* is comparable to the time spent in Briki. The time attributed to Kaffe in Figure 6.8 is larger because we categorize the start-up time and the time to load all class files as "Kaffe overhead."

7 Conclusions

7.1 Summary

We have designed new optimization techniques which speed programs up by improving their cache locality. We have implemented those techniques in optimizing compilers for C, Fortran and Java and performed experiments on standard benchmarks. Our experiments show that array transformations alone offer big speedups over existing techniques. Unified array and loop transformations offer further improvements in performance. Specific contributions of our work are:

- We have developed an algebraic framework which integrates representation of array mappings, array references and their locality properties. The framework and crucial new concepts of mapping vectors and stride vectors are presented in Chapter 2. We develop important theorems that allow a compiler to use array mappings without memory overhead and generate efficient code to access array elements. We also show how mapping vectors can be used for compact representation of banded matrices. Representing banded matrices within our framework enables the use of our optimizations for such arrays.

- We use array transformations to optimize *explicitly parallel programs*. Our optimizing compiler for SPMD programs offers dramatic improvements for programs running on cache-coherent shared-memory programs by reducing false sharing. The experimental results are discussed in Section 2.6.

- We demonstrate the use of array transformations for sequential Fortran programs by optimizing applications which cannot be sped up by other techniques. Furthermore, we show that there exist applications which cannot achieve their best performance when the compiler applies both loop and array transformations separately one after another (in any order). We show that an approach unifying both types of transformations is needed and propose such a unified technique which simultaneously considers both types of

transformations. We integrate existing frameworks for loop transformations (which represent loop optimizations with transformation matrices) with our array transformations. Chapter 3 presents both the theoretical framework and experimental results for unified optimizations.

- In Chapter 4 we address the issue of array linearization which for some applications prevents our compiler from performing array transformations. Array linearization has been identified before as an obstacle to some high-performance optimizations (e.g. it makes dependence tests difficult and inhibits efficient array distributions in HPF), but we are first to present an algorithm for recovering logical multi-dimensional structure from one-dimensional arrays. We also propose a representation of *non-rectangular* arrays. Non-rectangular arrays (e.g. *triangular* arrays) have been used frequently, but the programmer always had to manually manage such arrays in flat, linear arrays. We suggest that an explicit support for such arrays be added to high-level languages. But we also show, in Section 4.4, how a non-rectangular type can be automatically recognized in a programming language without explicit support for such types. We anticipate that such type information will be useful for parallelizing and optimizing existing Fortran programs.

- Array aliasing is an important issue which must be addressed to make array transformations possible. In Chapter 5 we define *array overlapping* as an extension to the existing notion of aliasing. We show how to identify overlapping, how to reduce it by *selective cloning* and how to unify types of all overlapping arrays. As an additional optimization enabled by array overlap analysis, we introduce *code structure recovery* which is analogous to array structure recovery discussed in Chapter 4. We implement all techniques discussed in Chapter 5 in our optimizing Fortran compiler and run experiments on applications which up to that moment have eluded locality optimizations.

- We recognize the importance of Just-In-Time (JIT) compilers which are used with the new technologies for mobile software, like Java, which have rapidly gained acceptance in the past year or two. We notice that most high-performance optimizations that have been traditionally used for compiling languages like Fortran are not very well suited to JIT compilation. Those traditional algorithms are relatively slow and consume significant machine resources. In Chapter 6 we look at the possibility of the application of array transformations for Java programs. To make our optimizations competitive, we design new, fast, approximate compiler algorithms. We present those algorithms in Section 6.4. We implement array transformations in a Java JIT compiler perform experiments to demonstrate that our new algorithms

do not use accuracy for typical, scientific applications. Also the overhead of our optimizations in the compilation speed is very small compared to other functions of a JIT compiler. Section 6.5 presents experimental results obtained with our compiler.

7.2 Future Work

Our work for off-line optimization of typical access pattern in scientific applications seems to be fairly complete within the framework considered by us. Nevertheless, the opportunities for future research are abundant in the directions not followed in this dissertation.

Further work on non-rectangular arrays seems to be very promising by enabling optimizations which are not possible with any of the existing techniques. We have started the work by developing the algorithm for identifying and representing such arrays. However, we do not know how to use this type information effectively and how much support for such arrays (if any) is needed in programming languages.

Banded matrices are a very interesting problem. Important applications exist which use such arrays and further investigation into optimizing those programs is needed. We believe that our work (with previous work by Li presented in [47]) can be used to optimize such programs, but the issue of effective recognition of banded arrays remains open.

We believe that Just-In-Time compilation is the most interesting new direction stemming from our research. JIT compilers are becoming very widely used, yet optimizations techniques for those compilers are in their infancy. We would like to continue work on those issues in the future.

Bibliography

[1] A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and Language-Independent Mobile Programs. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 127–136, Philadelphia, PA, May 1996.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[3] J. M. Anderson and M. S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 112–125, Albuquerque, NM, June 1993.

[4] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and Computation Transformations for Multiprocessors. In *Proceedings of the Fifth ACM Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[5] B. Baker. An Algorithm for Structuring Flowgraphs. *Journal of the ACM*, 24(1):98–120, January 1977.

[6] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An Interactive Environment for Data Partitioning and Distribution. In *Proceedings of the Fifth Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[7] D. Bau, I. Kodukula, V. Kotlyar, K. Pingali, and P. Stodghill. Solving Alignment using Elementary Linear Algebra. In *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[8] M. Berry and others. The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, 3(3):9–40, Fall 1989.

[9] R. Bianchini and T. J. LeBlanc. Software Caching on Cache-Coherent Multiprocessors. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pages 521–526, Dallas, TX, December 1992.

[10] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The Next Generation in Parallelizing Compilers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing: 7th International Workshop*, volume 892 of Lecture Notes in Computer Science. Springer-Verlag, Berlin/Heidelberg, 1995.

[11] P. Budnik and D. J. Kuck. The Organization and Use of Parallel Memories. *IEEE Transactions on Computers*, C-20(12):1566–1569, December 1971.

[12] D. Callahan and A. Porterfield. Data Cache Performance of Supercomputer Applications. In *Supercomputer '90*, 1990.

[13] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler Optimizations for Improving Data Locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, San Jose, CA, October 1994.

[14] C. Chambers, J. Dean, and D. Grove. Whole-Program Optimization of Object-Oriented Languages. TR UW-CSE-96-06-02, Department of Computer Science and Engineering, University of Washington, 1996.

[15] R. Chandra, D.-K. Chen, R. Cox, D. E. Maydan, N. Nedeljković, and J. M. Anderson. Data Distribution Support on Distributed Shared Memory Multiprocessors. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, June 1997.

[16] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. Automatic Array Alignment in Data-Parallel Programs. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages*, Charleston, SC, January 1993.

[17] M. Cierniak and W. Li. Unifying Data and Control Transformations for Distributed Shared-Memory Machines. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[18] M. Cierniak and W. Li. Briki: A Flexible Java Compiler. TR 621, Computer Science Department, University of Rochester, May 1996.

[19] M. Cierniak and W. Li. Recovering Logical Structures of Data. In C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing: Eighth International Workshop*, volume 1033 of Lecture Notes in Computer Science. Springer-Verlag, Berlin/Heidelberg, 1996.

[20] M. Cierniak and W. Li. Briki: an Optimizing Java Compiler. In *Proceedings of the IEEE COMPCON '97*, San Jose, CA, February 1997.

[21] M. Cierniak and W. Li. Optimizing Java Bytecodes. *Concurrency: Practice and Experience*, 9(6):427–444, June 1997.

[22] M. Cierniak and W. Li. Interprocedural Array Remapping. To appear in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '97)*, San Francisco, CA, November 1997.

[23] M. Cierniak and W. Li. Just-In-Time Optimizations for High-Performance Java Programs. To appear in *Concurrency: Practice and Experience*, 1998.

[24] C. Cifuentes. Structuring Decompiled Graphs. In *Proceedings of the International Conference on Compiler Construction*, volume 1060 of Lecture Notes in Computer Science, pages 91–105. Springer-Verlag, Berlin/Heidelberg, April 1996.

[25] K. D. Cooper and K. Kennedy. Fast Interprocedural Alias Analysis. In *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages*, pages 49–59, Austin, TX, January 1989.

[26] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proceedings of the Twentieth International Symposium on Computer Architecture*, pages 88–97, San Diego, CA, May 1993.

[27] S. J. Eggers and T. E. Jeremiassen. Eliminating False Sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages I:377–381, St. Charles, IL, August 1991.

[28] C. Eisenbeis, W. Jalby, D. Windheiser, and F. Bodin. A Strategy for Array Management in Local Memory. In *Proceedings of the Third Annual Workshop on Languages and Compilers*, August 1990.

[29] J. M. Frailong, W. Jalby, and J. Lenfant. XOR-Schemes: A Flexible Data Organization in Parallel Memories. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 276–283, St. Charles, IL, August 1983.

[30] D. Gannon, W. Jalby, and K. Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformations. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[31] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley Longman, Inc., Reading, MA, 1996.

[32] M. Gupta and P. Banerjee. Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[33] M. Gupta and P. Banerjee. PARADIGM: A Compiler for Automatic Data Distribution on Multicomputers. In *1993 ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[34] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[35] D. E. Hudak and S. G. Abraham. Compiler Techniques for Data Partitioning of Sequentially Iterated Parallel Loops. In *1990 ACM International Conference on Supercomputing*, pages 187–200, Amsterdam, The Netherlands, June 1990. In *ACM SIGARCH Computer Architecture News 18*:3.

[36] T. E. Jeremiassen and S. J. Eggers. Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations. In *Proceedings of the Fifth ACM Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[37] Y.-J. Ju and H. Dietz. Reduction of Cache Coherence Overhead by Compiler Data Layout and Loop Transformations. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 344–358. Springer-Verlag, August 1991. Fourth International Workshop, Portland, OR.

[38] K. Kennedy and U. Kremer. Automatic Data Layout for High Performance Fortran. In *Proceedings Supercomputing '95*, San Diego, CA, December 1995.

[39] K. Kim and V. K. P. Kumar. Parallel Memory Systems for Image Processing. In *Proceedings of the 1989 Conference on Computer Vision and Pattern Recognition (CVPR '89)*, pages 654–659, 1989.

[40] K. Knobe, J. Lukas, and G. Steele. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8, February 1990.

[41] D. H. Lawrie. Access and Alignment of Data in Array Processors. *IEEE Transactions on Computers*, C-24(12):1145–1155, December 1975.

[42] S. Leung and J. Zahorjan. Optimizing Data Locality by Array Restructuring. TR-95-09-01, Department of Computer Science and Engineering, University of Washington, September 1995.

[43] J. Li and M. Chen. Index Domain Alignment: Minimizing Cost of Cross-Referencing Between Distributed Arrays. In *Proceedings of the Third Symposium on Frontiers of Massively Parallel Computation*, College Park, Maryland, October 1990.

[44] X. Li. Parallel Algorithms for Hierarchical Clustering and Cluster Validity. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(11):1088–1092, November 1990.

[45] W. Li and K. Pingali. Access Normalization: Loop Restructuring for NUMA Compilers. *ACM Transactions on Computer Systems*, 11(4):353–375, November 1993.

[46] W. Li and K. Pingali. A Singular Loop Transformation Framework Based on Non-Singular Matrices. *International Journal of Parallel Programming*, 22(2), April 1994.

[47] W. Li. Compiler Cache Optimizations for Banded Matrix Problems. In *1995 ACM International Conference on Supercomputing*, pages 21–30, Barcelona, Spain, July 1995.

[48] L. M. Liebrock and K. Kennedy. Parallelization of Linearized Applications in Fortran D. In *Proceedings of the Eighth International Parallel Processing Symposium*, Cancun, Mexico, April 1994.

[49] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Longman, Inc., Reading, MA, 1996.

[50] Z. Liu, X. Li, and J.-H. You. On Storage Schemes for Parallel Array Access. In *1992 ACM International Conference on Supercomputing*, pages 282–291, Washington, DC, July 1992.

[51] S. Lucco, O. Sharp, and R. Wahbe. Omniware: A Universal Substrate for Web Programming. In *Proceedings of the Fourth International World Wide Web Conference*, Boston, MA, December 1995. Available at `http://www.w3.org/pub/Conferences/WWW4/Papers/165`.

[52] Lucent Technologies Inc. The Limbo Programming Language. May 1997. Available at `http://inferno.lucent.com/inferno/limbo.html`.

[53] V. Maslov. Delinearization: An Efficient Way to Break Multiloop Dependence Equations. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992.

[54] V. Maslov and W. Pugh. Simplifying Polynomial Constraints Over Integers to Make Dependence Analysis More Precise. In *International Conference on Parallel and Vector Processing (CONPAR '94 — VAPP VI)*, Linz, Austria, September 1994.

[55] A. Nagurney, C. F. Nicholson, and P. M. Bishop. Spatial Price Equilibrium Models with Discriminatory *ad valorem* Tariffs: Formulation and Comparative Computation Using Variational Inequalities. In J. C. J. M. van den Bergh, P. Nijkamp, and P. Rietveld, editors, *Recent Advances in Spatial Equilibrium Modeling: Methodology and Applications*. Springer-Verlag, Heidelberg, 1993.

[56] J. Plevyak and A. A. Chien. Type Directed Cloning for Object-Oriented Programs. In C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing: Eighth International Workshop*, volume 1033 of Lecture Notes in Computer Science. Springer-Verlag, Berlin/Heidelberg, 1996.

[57] D. Pountain. Parallel Course. In *BYTE*, July 1994. Available at `http://www.byte.com/art/9407/sec6/art1.htm`.

[58] J. Ramanujam and P. Sadayappan. Compile-Time Techniques for Data Distribution in Distributed Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, October 1991.

[59] R. Sass and M. Mutka. Enabling Unimodular Transformations. In *Proceedings, Supercomputing '94*, Washington, DC, November 1994.

[60] Z. Shen, Z. Li, and P.-C. Yew. An Empirical Study on Array Subscripts and Data Dependencies. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages II:145–152, St. Charles, IL, August 1989.

[61] Silicon Graphics, Inc. MIPSPro Fortran Programmer's Guide. 1996.

[62] Standard Performance Evaluation Corporation. SPEC CPU92 Benchmarks. April 1996. Available at `http://open.specbench.org/osg/cpu92/`.

[63] Standard Performance Evaluation Corporation. SPEC CPU95 Benchmarks. March 1997. Available at `http://open.specbench.org/osg/cpu95/`.

[64] Tao Systems Ltd. The Tao Operating System. December 1996. Available at `http://www.tao.co.uk/taos.htm`.

[65] H. van Vliet. Mocha the Java decompiler. November 1996. Available at `http://web.inter.nl.net/users/H.P.van.Vliet/mocha.htm`.

[66] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, pages 201–207, Durham, NC, January – February 1994.

[67] H. A. G. Wijshoff. *Data Organization in Parallel Computers*. Kluwer Academic Publishers, Boston, MA, 1989.

[68] T. Wilkinson. Kaffe v0.8.3 — A Free Virtual Machine to Run Java Code. March 1997. Available at `http://www.kaffe.org/`.

[69] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, Canada, June 1991.

[70] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, Reading, MA, 1996.

[71] World Wide Web Consortium. Mobile Code. May 1997. Available at `http://www.w3.org/pub/WWW/MobileCode`.