This is to certify that the

dissertation entitled

MATRIX-BASED RESTRUCTURING COMPILERS

presented by

Ron Sass

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Computer Science
and Engineering

Thomas W. Mutka

Major professor

Matt Mutka

Date February 26, 1999

MATRIX-BASED RESTRUCTURING COMPILERS

By

Ron Sass

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

1999

ABSTRACT

MATRIX-BASED RESTRUCTURING COMPILERS

By

Ron Sass

Scientists and engineers requiring high-performance computing systems have faced the same problem for the last thirty years: to get more performance, mainly in the form of reduced time-to-completion, software has to be re-coded with every major hardware and system advancement. Restructuring compilers help address this fundamental problem by extensively analyzing the users' code and mapping it to a high-performance machine, giving the user more hardware independence. Matrix-based restructuring compilers accomplish this by using transformation frameworks. In the matrix-based frameworks, vectors represent the dependence between loop iterations and matrices represent the transformations. By using a measure of performance in the abstract framework, algorithms can be written for these frameworks that — in polynomial time — find an ordering of transformations that maximizes the performance metric. This is a significant improvement over brute-force algorithms used prior to this development.

This dissertations aims to improve restructuring compilers that use matrix-based frameworks. Namely, we address a serious limitation of matrix-based frameworks that prevents

them from transforming imperfectly nested loops. We offer two approaches to address this limitation. First, we develop a new loop classification scheme which we use to analyze 25,000 lines of scientific kernel and benchmark programs. As a result of this analysis, we show that the composition of two existing, slightly-modified transformations can be used to create perfectly nested loops from imperfectly nested loops. We also present a technique, called the *phase method*, which allows matrix-based transformations to operate directly on doubly-nested, imperfect loops.

Also in support of restructuring compilers, we look at the problem of mapping applications to a relatively new high-performance architecture, networks of workstations (NOWs). Commodity parts used in NOWs make the cost/performance ratio very attractive, but also complicate the development of a suitable performance measure for matrix-based transformations. In this dissertation we develop and test a model that uses feedback from calibration runs on the hardware to automatically determine such a measure. To test its effectiveness we consider both statistical evaluation and the success of a practical application (determining the number of useful processors in a parallel program).

As high-performance computers come to rely more on parallelism for increases in performance, the role of the restructuring compiler becomes more crucial. This dissertation, and the investigations described within, propose improvements to the design of the modern restructuring compiler.

ACKNOWLEDGMENTS

Many people have contributed in many ways to bringing this dissertation to fruition. I am indebted to my very patient advisor, Dr. Matt Mutka, for his guidance on all matters of my scholarly career. Even though I have not unfailingly followed his advice, I have come to regret the times I didn't. I would also like to thank my committee. I was very fortunate to have an interested, helpful, and very competent committee. I would like to thank the Department of Computer Science and Engineering for employing me throughout my degree and for providing me with a truly outstanding education.

There are many students who have contributed indirectly to this work. By their friendship and goodwill, I have kept in good spirits for my duration at MSU. This includes Joe Sharnowski, my roommate at Owen Graduate Center, who always kept me on track that first quarter, recommended me for the systems manager job, and who welcomed me into his group of friends. Goodness gracious! I'm done! Edgar Kalns, Marie-Pierre Jolly, Christian Trefftz, and the rest of the "lunch crowd" were delightful company. I never imagined such stimulating discussions over lunch. Lynn Kubinec showed me, perhaps unknowingly, that great advantages of a well-organized, focused work habits. Others suggested that I "stay focused" but Lynn set an example. I also appreciated her frequent cultural outings. For less cultural outings, I have to thank the Llama Pad for hosting numerous Halloween and hot-tub parties. Steve Turner has been supportive as both a technical and a nontechnical partner; I hope we can continue in both partnerships. Stephen Wagner has also been a true

friend and has seen me through some rough times. The students and Rick Erickson at the Wesley Foundation have provided me with programs, companionship, and guidance while I was in school.

I owe a special word of gratitude to Kelly S. Mack. We started this together and you contributed in so many ways I cannot record them all here. I will forever remember the sacrifices you made for me.

Some people have contributed directly towards this dissertation. Martha Sorbet deserves a great deal of appreciation. Not only was she a source of encouragement, but she stayed up many late nights with me and learned vi and LaTeX just so that she could help type and edit my papers and dissertation. Thank-you. My father, Joe Sass, also contributed directly to this work. He read the dissertation several times and made numerous suggestions. Much of what I know about written communication, I owe to him.

Finally, I'd like to thank my friend and colleague, Barb Gannod. From start to finish, you were my closest friend throughout my time in graduate school.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

> "*Raios cubicos*!" he says, with a vengeance. Cube roots! ... It's hard to find a more difficult fundamental problem in arithmetic ... How did the customer [Feynman] beat the abacus? The number was 1729.03. I happened to know that a cubic foot contains 1728 cubic inches, so the answer is a tiny bit more than 12. The excess, 1.03, is only one part in nearly 2000, and I had learned in calculus that for small fractions, the cube root's excess is one-third of the number's excess. So all I had to do is find the fraction 1/1728, and multiply by 4 (divide by 3 and multiply by 12). So I was able to pull out a whole lot of digits that way.
>
> **Richard Feynman** [27]

Engineers and scientists use a number of tools to perform computations. For some complicated problems, high performance computers (HPCs) are needed because the time to complete the computation can take hours or days. This slow turn-around time diminishes the value of the results. A large decrease in execution time justifies the cost of HPCs and the development of sophisticated tools. One of the tools in this arena is the restructuring compiler. A *restructuring* compiler is a compiler that not only translates source to object code but also improves the performance of the application by analysis that is beyond what

is necessary for direct semantic translation. Frequently, the restructuring compiler is designed to improve the performance by reorganizing the code to find parallelism and to place data in memory more effectively. Matrix-Based restructuring compilers implement the code-improving transformation framework by modeling applications and transformations as mathematical structures such as vectors and matrices. It has been shown by Banerjee [9] that matrix-based frameworks have the ability to use the same mechanisms to compile for different goals.

Although the source code could be written with the code improvements explicitly noted, restructuring compilers are still very important tools because they address three concerns. First, manually finding the parallelism in an algorithm is initially a rote exercise for the programmer but the problem becomes increasingly difficult after the easy cases have been discovered. Second, a significant body of code already exists in serial notation (commonly referred to as "dusty deck" code). Finally, high performance computers are changing rapidly and good object code for one machine may not be good for subsequent machines. Restructuring compilers address all three of these concerns because the human effort involved in each concern is usually much more expensive than using an automated tool.

The machinery of HPCs has changed continually over the years. While the demand for higher performance has not diminished, many vendors that have manufactured HPCs in the last fifteen years have either scaled back the HPC division or failed. While the underlying reasons for these failures are still being debated (as noted by Kuck [42]) the problem on the surface is one of economics. First, the hardware cost was too expensive to support continued sales. And second, expensive hardware hurts the cost/performance ratio of the HPC.

One place where parallelism — the heart of many HPCs — *has* succeeded is at the instruction level [16]. The success of the workstation microprocessor and its instruction-level parallelism has fueled interest in HPC systems built from workstations and local area networks. Such a system is often called a *network of workstations* (NoW) when all of the individual workstations in the system are equivalent or a *cluster computer* when the workstations vary in power or architecture. [1] For both, the chief attraction is the (potential) cost/performance ratio. Central to the NoW's cost/performance advantage is its use of off-the-shelf components. By using commodity components — microprocessors, operating systems, and networks — the cost is kept low while still taking advantage of the advances in microprocessor instruction-level parallelism.

Compiling for NoWs is especially interesting because of the unique problems it presents. The off-the-shelf components, by their very nature, are general-purpose. Different vendors' components can be interchanged and intermixed. The compiler does not know what to expect of the hardware. The performance successes of the microprocessor are outpacing the advances in networks, which means the compiler cannot rely on a constant performance ratio between the two. Furthermore, local area networks today are designed for video and audio traffic; it is likely that future network improvements will continue to address multimedia demands while parallel computing issues — such as reducing the message startup costs and latency — are likely to be of secondary concern.

Thus, our thesis is that matrix-based restructuring compilers, operating on two levels, are instrumental in the cost-effective use of NoWs as HPCs. Specifically, at one level, the

[1] Our distinction is not universal. For example, at UC Berkeley they refer to one apparatus "for cluster computing" as the Berkeley NOW [48].

compiler needs to find sufficient fine-grain parallelism to fully utilize the modern microprocessor. On a second level, the compiler needs to find coarse-grain parallelism under the unique conditions of the NoW environment. To advance this goal, this dissertation focuses on two concerns. First, we address the limitation of perfectly nested loops, which was a general limitation of matrix-based restructuring compilers. And second, we describe how a restructuring compiler can naturally adapt to the variable nature of NoWs. Below, we discuss these two aspects of the dissertation's work in more detail, describe the fundamental problems, and state our specific contributions.

Imperfect Loop Nests An early concern expressed by the research community with matrix-based transformation frameworks was that the framework was only applicable to perfectly nested loops (see Banerjee [10]), such as the loop nest below.

```
DO I=1,N
  DO J=1,M
    X(I,J)=A(I,J)*V(J)
  END DO
END DO
```

For performance reasons, a restructuring compiler may transform this loop in a number of ways. For reasons described in Chapter 2, matrix-based restructuring compilers are very effective for this loop nest structure and several algorithms exist for matrix-based frameworks that exploit a particular HPC's architectural features. Occasionally though, loops are not perfectly nested. The following loop is *imperfectly* nested:

```
DO K=1,N
  A(K,K)=A(K,K)**0.5
  DO I=K+1,N
    A(I,K)=A(I,K)/A(K,K)
    DO J=K+1,I
      A(I,J)=A(I,J)-A(I,K)*A(J,K)
    END DO
  END DO
END DO
```

Prior to our work, matrix-based restructuring compilers always assumed perfectly nested loops. If the loop was not perfectly nested, it was not parallelized. Even for general restructuring compilers, there have only been a few techniques developed over the years. In [68], Wolf and Lam mention that general loops remain an open problem (for the matrix-based approach). In the technical report [37], Kelly and Pugh partially address the problem of imperfectly nested loops, but their technique does not embrace the loop transformation ordering abilities of the matrix-based approach. The author was first aware of the problem when Banerjee posed it [10].

Our first contribution is to analyze and classify the types of imperfectly nested loops for a large body of FORTRAN codes. We found a great deal of overlap in most of the (non-matrix-based) transformations developed previously to convert imperfectly nested loops into perfectly nested loops. We call the loops commonly transformed, the class of apperfectly nested loops. Another contribution, shown in Chapter 3, is a demonstration of how all matrix-based frameworks can be augmented to handle apperfect loops. This establishes that matrix-based techniques are not, in general, less powerful because the original theory did not include imperfectly nested loops.

Furthermore, we show that there is a simple procedure for extending the unimodular framework to include imperfectly nested loops, when the goal is fine-grain parallelism. This technique, called the phase method, is described in Chapter 4.

Restructuring for NoWs Returning to the second aspect, all restructuring compiler frameworks share two common features that distinguish them from other restructuring compilers. All frameworks are based on some mathematical model of loop nests (and transformations) and some implicit (or explicit) performance estimator. The latter is usually expressed as some goal, such as, "maximum parallelism transformation" or "communication-free transformation." In these cases, the goal represents some general understanding of performance for a particular architecture. The problem with the performance estimators mentioned above is that they are too general for NoWs. While maximum parallelism may be the best solution for some applications, it is certainly not the best solution for all. The other extreme, that of no communication (at the cost of parallelism), is also too general. Furthermore, both assume that the architecture is generally fixed and has few options available. The problem is more complex when we desire to compile for a NoW built from commodity processors, off-the-shelf operating systems, and third-party networks. It is difficult or impossible to determine how the different manufacturers' components will effect message start-up times, latency, and possible contention. Distributed memory multicomputers with dedicated interconnection networks do not have this variability.

We have investigated the use of a new performance estimator to be incorporated into matrix-based transformation frameworks in Chapter 5. This approach is especially suitable for NoWs. Another contribution of this dissertation is to show that NoWs do have

properties that can be parameterized and used in a performance estimator for matrix-based restructuring compilers. Our procedure uses a calibration step to determine the properties and then uses the results as a performance metric in subsequent compilations. With a better performance estimator, a more precise goal for the framework can be defined.

Finally, we use this measure to solve one specific problem. Figure 1.1 shows a speedup curve that is familiar to researchers who study parallel processing. It embodies the general rule that performance will improve as the number of processors increase — up to a point. After that, communication costs cause the performance to decrease. One use of the performance estimator is to identify the *shape* of the performance curve prior to execution. The final contribution of this work is to show that, after calibration, our metric predicts when communication costs will dominate and additional processors are unneeded or will hinder performance. The metric chooses the best number of workstations to use on a specific problem.

One application of the previous solution is in a High Performance Fortran (HPF) compiler, where the user is responsible for choosing the number of *virtual* processors and the distribution of the data, but the compiler must decide how many *physical* processors to use. For a NoW, the choice "all of the available workstations" may not be an efficient use of the facilities nor an ideal choice (as Figure 1.1 indicates). Also, the compiler still needs to choose the number of physical processors for non-HPF compilers, where the compiler is responsible for the data distribution and translation.

The remainder of this dissertation is organized as follows. In Chapter 2, we discuss the basics of restructuring compilers. We give a brief survey of developments, a description

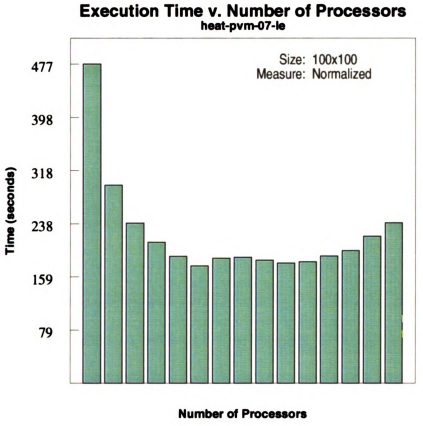

Figure 1.1: typical performance curve for a small data-parallel application

of the usual loop transformations, and work related to this dissertation. In Chapter 3, we describe how to handle the class of apperfectly-nested loops in a matrix-based restructuring compiler. In Chapter 4, we describe how to use the phase method to handle imperfectly-nested loops for fine-grain parallelism. In Chapter 5, we discuss compiling for NoWs. We describe our data-parallel testbed, the development of a cost function for NoWs, and the results of using the cost function to determine the critical point of speedup curves, *a priori*. Conclusions and future work are presented in Chapter 6.

Chapter 2

Restructuring Compiler Basics

Good order
is the foundation
of all good things.

Edmund Burke [17]

In this chapter we discuss the basics of restructuring compilers. We define the major types of restructuring compilers, present a brief survey of restructuring compiler advances, and describe the loop transformations used throughout the dissertation. Fundamental to all framework-based restructuring compilers is their ability to order the transformations. This is the key technical point of the chapter.

2.1 Restructuring Compilers

As mentioned in the introduction, a restructuring compiler is one that, besides translating the source code to object code, does additional analysis to improve the performance. It is

similar to an optimizing compiler because both improve the source code's performance for a particular architecture. But generally, a restructuring compiler does more. The restructuring compiler finds parallelism in large code structures and attends to the problem of accessing the memory hierarchy efficiently. The distinction between optimizing and restructuring compilers is informal and blurring as they converge, sharing technology and terms. Indeed, most optimizing compilers search for parallelism and change reference patterns to use the memory hierarchy efficiently. Likewise, restructuring compiler transformations are frequently called optimizations.[1]

There are three main groups of restructuring compilers, distinguished by how they order their transformations. Early designs allowed restructuring compilers to perform a large number of transformations. But each transformation was applied individually, one after the other. The intention was that, as new transformations are developed, they could be integrated easily. New transformations were being developed frequently so having a compiler that performed a static list of transformations was the most convenient research tool. A problem arises when researchers try to establish a "best" order for the transformations. One solution that we call the *ad hoc* approach, is to simply fix an order based on past experience. A second solution to the best-order problem is to generate a sequence of transformations and then evaluate the result based on some performance estimate, to determine if the transformation was profitable. Subsequently, the compiler generates another sequence and re-evaluates the result. This philosophy of *generate-and-test* is a brute-force approach to finding the best order. The third approach is to use a mathematical framework to model

[1]Frequently, researchers use "optimizations" in place of transformations. Although ubiquitous, only a few researchers are finding transformations that can be shown to be locally optimal. It is nearly impossible to prove that any transformation is globally optimal.

the transformations and then, based on the model, generate a single transformation. This group of compilers can be further subdivided into those that use matrices to represent transformations and those that manipulate data dependence equations directly. The drawback to the matrix-based approach is that currently only a few transformations can be modeled and only perfectly nested loops are transformed. An important advantage to using frameworks is that with the model, one can establish precise metrics. Polynomial-time algorithms can be designed to solve the best-order problem to optimize for particular goals. Finding the best order is further complicated when more than just parallelism is needed — for example, when compiling for improved parallelism and reduced communication.

The two subtypes of matrix-based restructuring compilers depend on what set of matrices are used to represent the transformations. The older unimodular matrices encode three loop transformations. The new non-singular matrices encode four loop transformations. The latter was developed to include more efficient use of the memory hierarchy.

The hierarchy of restructuring compilers just described is given in Figure 2.1. The leaves are examples of restructuring compiler projects of that type.

2.2 Survey of Developments

In this section we discuss the important developments that have led to unimodular transformations and other matrix-based transformations. Restructuring compiler developments came from a number of different sources. We begin with a brief history of vectorizing and restructuring compilers. Then we summarize specific works that have had a significant influence on matrix-based transformations.

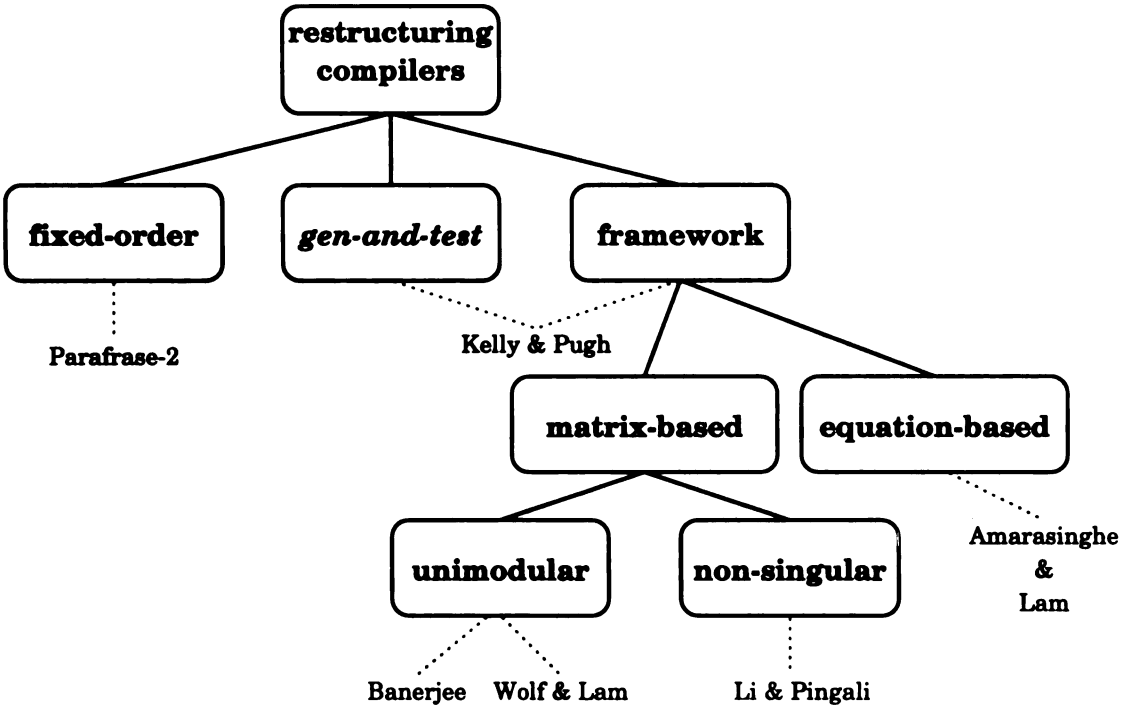

Figure 2.1: organization of restructuring compilers

The first restructuring compilers were designed for vector computers. Vector computers increase performance by pipelining a set of data through a functional unit; different stages of the computation of the function operate on different data in parallel. The compiler's main responsibility for vector computers was to make loops suitable for this type of computation. Thus converting singly-nested loops and the inner loop of a nest to a suitable form for the vector units was sufficient to fully utilize the architecture.

2.2.1 University of Illinois

We begin our survey with the work of David Kuck [41] and his students at the University of Illinois. With Kuck came data dependence analysis. Data dependence is the criterion we use to mark the start of restructuring compiler research. Prior to Kuck's work, researchers discussed compiler techniques to execute code in parallel, but none of the techniques were able to handle subscripted array references. The one exception is Leslie Lamport's oft-cited paper [43]. Lamport's *hyperplane method* accepts loops of a very specific form and generates parallel code, but it does not use the concept of data dependence. This is a very important exception because the hyperplane method (after data dependence was established) became the basis of the wavefront transformation which is fundamental to many matrix-based frameworks.

Data dependence, as mentioned above, is key to all the current restructuring compilers. The dependence relation is frequently represented pictorially as a directed graph. Internally, it is represented as a vector. By using this formalism, Kuck and his associates developed

algorithms that remove inter-statement dependences, such that statements within a basic block can be executed in parallel [41].

2.2.2 Rice University

At Rice University in Houston, Texas, Ken Kennedy and his research group [38, 39] also pioneered restructuring compiler techniques. Kennedy and his associates' work includes PFC (Parallel Fortran Converter) which is a FORTRAN-TO-FORTRAN compiler, a dependence browser (PTOOL), and ParaScope, an interactive parallel programming environment. Additionally, this research team developed FORTRAN D, which has strongly influenced the specification of HPF (High Performance Fortran). Similar to the work at Illinois, the researchers at Rice used an *ad hoc* approach, not a matrix-based approach.

Researchers have continued to spend considerable time investigating data dependence. It continues to attract the the attention of new researchers in an attempt to develop faster, more accurate analysis. For a treatise on dependence analysis, many of the researchers reference Banerjee [8]. However, we suggest Banerjee's book [11] for those interested in matrix-based transformations. For an excellent bibliography on dependence testing, see Pugh's Omega Test [56]. The dependence analysis in PFC is described in [31]. Another approach to performing dependence analysis is proposed by Maydan, Hennessy, and Lam in [49]. For investigators with little or no background in this field of research, we recommend Hans Zima's *Supercompilers for Parallel and Vector Computers* as a launching point [74].

The strategy of Kuck's compiler group was to perform source-to-source translations based on data dependence analysis. Code transformations are applied in a fixed order — the *ad hoc* approach. While some interactive aspects have been added recently, their approach does not easily include user feedback in future compilations. Wolfe's thesis and book *Optimizing Supercompilers for Supercomputers* [72] is a comprehensive summary of this approach. Parafrase-2, a research compiler, is another excellent representative example of this approach [55].

2.2.3 Systolic and VLSI Architectures

Research that targeted systolic computation and specialized (parallel) VLSI architectures was instrumental in the next development phase of matrix-based compilers. Starting with Lamport's hyperplane method, several researchers established a number of important results.

One of the first goals was, given a set of index points with a dependence relation, to find an independent partition of the set of index points. An index point is simply a vector that represents one iteration of a loop nest. An independent partition is a partition of the index set such that no dependences exists between any two subsets of the partition. With an independent partition, it is easy for the compiler to produce parallel threads of execution with no synchronization primitives except at the end. The independent partition problem was proposed in Padua's thesis [33]. He presents the Greatest Common Divisor solution in his thesis.

For the independent partition problem, a partition with larger cardinality is a better partition because it increases the parallelism. Peir and Cytron discuss a method called Minimum Distance in [54], which finds better solutions than the Greatest Common Divisor method but with added cost at compile time. Shang and Fortes present a solution in [62] that is a compromise between (compile-time) algorithmic complexity and good partitions. Although the researchers find an independent partition, they lacked a a technique to output the transformed code. D'Hollander's work also solves the independent partition problem and he includes a *labeling* algorithm which outputs the transformed code [21]. However, his resulting code is unnecessarily complex and carries extra run-time overhead. The final missing piece is in Ancourt and Irigoin's paper [5], which (discussed below) includes an elegant solution for outputting the transformed code.

2.2.4 Transformation Frameworks

The next group of developments in restructuring compilers came as researchers started to develop frameworks. This work emenates directly from the research described in the previous section. The main distinction is that the papers in the previous section targeted primarily systolic-type machines or specialized VLSI architectures while the papers in this section target multiprocessor machines.

"Supernode Partitioning" by Irigoin and Triolet was one of the early papers and is of great significance [36]. In supernode partitioning, the authors describe algorithms that cluster nodes (the authors' term for index points) into supernodes, where supernodes have certain desirable properties. Two such properties include nodes that may be executed in

parallel and nodes that exploit the memory hierarchy. This is the start of a systematic form of loop tiling. Irigoin and Triolet show that for any loop nest, transformations exist that create a loop nest with (at most) one sequential outer loop nest and one or more inner loops that execute in parallel. They sketch an algorithm that outputs transformed code for doubly nested loops.

In [5], Ancourt and Irigoin expand the work by Irigoin and Triolet [36] and discuss how to solve the problem of outputting simple, transformed loop nests, a problem which a number of previous papers ignored. Even as recently as 1992, D'Hollander's solution to the problem of outputting loop nests [21] was awkward because it introduced extra loops. The elegant solution shown in [5] is based on Fourier's pair-wise elimination.[2] The new solution produces new loop bounds based on the transformed iteration space. Missing from both [5] and [36] is a decision algorithm that finds the transformation matrix.

In the same time period that Ancourt and Irigoin were developing their ideas, Banerjee was devising a unified theory to represent three transformations [9]. This theory contains the results of the work presented above (in Irigoin and Triolet and in Ancourt and Irigoin) and also expands the theory to allow compiler designers to set performance measures and develop algorithms to optimize the transformation matrix. Banerjee presents this theory for doubly-nested loops, but later extends it in [11] and [12]. Banerjee also requires that the loop nest's dependences must all be uniform (that is, the set of distance vectors must be finite). Banerjee, for example, offered two algorithms in [9]. One algorithm maximized fine-grain parallelism; the second one algorithm maximized course-grain parallelism. The algorithms correspond to two different (reasonable) approximations of performance on gen-

[2]The solution is also known as Fourier-Motzkin Elimination and other names (see Schrijver [61]).

eral parallel machines, which is valuable for demonstration. But they are too general in practice because they ignore communication costs and other important factors.

In the same proceedings, Wolf and Lam [69] present a similar theory with the same benefit: a polynomial-time algorithm can be developed to maximize architecture-specific goals. They also present example algorithms that determine the transformation matrix. An important difference is that their framework is not limited to uniform dependences and their work discusses the theory for arbitrarily-deep nested loops. Wolf and Lam also discuss the theory in terms of loop tiling. One of their primary concerns is optimizing memory references for the memory hierarchy of a given architecture.

Both of these papers model transformations with unimodular matrices. Their models are limited to perfectly nested loops.

A number of researchers have extended the work of Banerjee and the work of Wolf and Lam. In [44] and [45], Li and Pingali discuss a framework based on non-singular matrices that provide an additional transformation, which is useful in NUMA architectures. The framework also eases the computation of the transformation matrix because it has fewer restrictions.

Other matrix-based approaches include an extension proposed by Ayguadé and Torres [7], which is similar to Li and Pingali [44]. Kelly and Pugh push for a generate-and-test decision algorithm with an artificial intelligence approach to reduce the search space [37]. In [47], Lu presents a formal transformation framework based on affine functions called "loop transformers." In addition to the usual three transformations, Lu includes statement reordering. Although Lu does not model the transformations with matrices, his ideas are similar to the matrix-based approach.

Chesney [19] used a formal approach to adapt matrix-based frameworks to support other transformations, such as loop distribution. His framework, however, requires more mathematical structures than just vectors and matrices. This complicates the task of finding a decision algorithm needed to order the transformations and no decision algorithm has yet been found for this system.

2.2.5 Data-Parallel Compiler Developments

Communication costs are extremely important. Even on specialized parallel computing hardware, the communication issue has received detailed attention. The earliest FORTRAN D implementations relied on compiler transformations to reduce and hide communication overhead [34]. Other researchers, such as Lim [46] and Huang [35], have considered communication so troublesome that they chose goals to address communication first and parallelism second. Both are communication-free transformations. The works by Amarasinghe [4] and Anderson [6] describe more advanced frameworks to address the critical issue of communication costs in a distributed memory machine. Other HPF-related papers, such as the work by Gupta and Schonberg [32], describe communication analysis and novel techniques for reducing communication by adjusting the protocol. The technique described by Garza-Salazar and Böhm *recompute* values to avoid communicating them [30].

2.3 Loop Nest Transformations

It is well known that programs tend to spend most of their execution time in a small portion of the code.[3] Execution time is not proportional to the amount of code because programs can *loop* and re-execute a portion of the code. Most restructuring compilers exploit this property by concentrating their search for parallelism in loops.

In this section, we introduce loop transformations. Loop transformations are the restructuring compiler's primary means of finding parallelism in serial code. This section is not intended to be a comprehensive list of transformations. Instead, we provide an introduction to our notation and definitions (2.3.1), concentrate on loop transformations that are important to matrix-based restructuring compilers (2.3.2), introduce several non-matrix-based transformations that are referenced in the dissertation (2.3.3), and discuss existing performance metrics (2.3.4).

2.3.1 Preliminaries

For scalar computers, the order of execution comes from the relative order of the statements as they appear in the source file (the *lexical* order of the statements) and the language prescribed sequential order of a loop (the *semantics* of the loop). A restructuring compiler looks at other orderings that produce identical results (a *legal* ordering) yet increase the amount of parallelism. In this section, we define a perfect loop nest and develop the concepts of data dependence and iteration space.

[3]This is commonly known as the 90/10 Locality Rule. See Hennessy and Patterson's work [53] for more information about temporal locality while executing code.

An explicit loop statement in a language may occur by itself or nested within the scope of another as shown below.

```
DO I=1,N
  DO J=1,M
    S1
  END DO
END DO
```

A loop nest is *perfect* if the loop's body meets one of two conditions: either (I) the body consists of a sequence of non-loop statements with a single-entry point and single-exit point (SESE) or (II) the body consists of exactly one perfect loop nest. This definition agrees with Abu-Sufah's definition of basic loops [1] and with the concept of perfectly nested loops used by Banerjee [9], Wolfe [72], and others. "Tightly-nested" has also been used by Lamport to describe perfect loop nests [43].

Symbolically, we can describe any perfect loop nest with the following notation. Given a nest of p loops, we have p lower bounds (denoted as $\mathtt{lb}_1$, $\mathtt{lb}_2$, ..., $\mathtt{lb}_p$), p upper bounds (denoted as $\mathtt{ub}_1$, $\mathtt{ub}_2$, ..., $\mathtt{ub}_p$), and p index variables (denoted as $\mathtt{i}_1$, $\mathtt{i}_2$, ..., $\mathtt{i}_p$). The bounds $\mathtt{lb}_1$,$\mathtt{ub}_1$ refer to the outer loop and $\mathtt{lb}_p$,$\mathtt{ub}_p$ refer to the inner loop. The value of the loop indices at any given iteration is stored in a vector (of length p) called an index point $I = [\mathtt{i}_1, \mathtt{i}_2, ..., \mathtt{i}_p]$, where $\mathtt{i}_1$ and $\mathtt{i}_p$ refer to the outer- and inner-most loop, respectively. The set of index points for all iterations of a nest is called the *iteration space*. For any given nest of loops there is a prescribed order for executing the index points, namely the serial (or sequential) order. For perfectly nested loops, this matches a lexicographical ordering of the index points.

Although the serial ordering is the execution order assumed by the programmer, the restructuring compiler — based on dependences between the index points — considers other orderings that produce semantically equivalent results. A *dependence* exists between two index points I_1 and I_2 when both use the same memory location and at least one index point writes to it. (Usually, dependences are further classified based on which index point is writing to the location, but we do not need that distinction here.) Assuming I_1 appears before I_2, lexicographically, and there is a dependence between the two, we can define a (dependence) distance vector as $D = I_2 - I_1$. Furthermore, we can define a direction vector corresponding to a given distance vector by applying the sign[4] function to each element (that is, the corresponding direction vector of the distance vector $[4, -3]$ is $[1, -1]$).

For a significant number of problems (which includes a number of linear algebra codes and all systolic array algorithms as described by Wolf and Lam [68]), there is a uniform pattern to all the distance vectors in the iteration space. Given a uniform pattern, the union of all of the distance vectors form a finite set — the dependence distance set. This set and the loop nest are the input to matrix-based transformation algorithms.

Finally, we define a dependence graph. A dependence graph is constructed by letting the index points be the vertices of the graph and the dependences be directed edges. A directed edge from I_1 to I_2 means that there is dependence from I_1 to I_2. To aid the reader, we always display dependence graphs in a p-dimensional Cartesian coordinate system with each vertex located such that its index point is its coordinate.

---

[4]The sign function is $sig(x) = \begin{cases} -1 & \text{if x is negative} \\ 0 & \text{if x is zero} \\ +1 & \text{if x is positive} \end{cases}$.

2.3.2 Matrix-Based Transformations

Banerjee [9] developed a unified, matrix-based theory by showing that three important loop transformations can be modeled by unimodular matrices. This allows us to approach specific problems with a rigorous theory to maximize goals in lieu of *ad hoc* techniques or the "generate-and-test" approach. It is our intention in this section to provide a brief overview of Banerjee's theory and other matrix-based transformations.

Unimodular matrices have several beneficial properties. They are integer matrices and have a determinant of ± 1. They are closed under matrix multiplication. For each of the three elementary loop transformations, which include permutations, skewing, and reversal, there is an elementary unimodular matrix that represents each transformation. Furthermore, it can be shown that every unimodular matrix corresponds to a (finite) sequence of elementary transformations and, conversely, that every (finite) sequence of elementary transformations can be represented by a unimodular matrix.

If I represents any $n \times n$ identity matrix, we can form the three elementary transformations easily. Loop permutation interchanges two DO loops in the nest; the corresponding unimodular matrix interchanges two rows of I. Loop skewing effectively maps an index point $[i_1, i_2, ..., i_q, ..., i_r, ..., i_p]$ to $[i_1, i_2, ..., i_q, ..., i_r + \mu i_q, ..., i_p]$ where μ is the skewing factor and loop r is being skewed with respect to loop q. This is accomplished in a unimodular matrix by replacing a zero in I with μ. Loop reversal switches the lower and upper bounds of a single loop so that the order of the iterations is reversed. This is accomplished by negating a diagonal element of I.

The final step is to transform the loop limits to output the new DO statements based on the transformation matrix. The loop limits are mapped to new values (based on the transformation matrix) and any expressions that use the new index variables inside the loop are modified to map back to the original values (using the inverse of the transformation matrix). These steps are critical to work presented here but the details are unimportant. The interested reader is directed to the work by Banerjee [9] for an introduction and Banerjee's book [11] for extensive coverage of the topic.

One of the example algorithms that Banerjee proposes [9] finds transformations that are suited for fine-grain parallel architectures. He proves that this algorithm guarantees that the transformed loop nest is legal and

- the inner loop can be executed in parallel, and
- the number of iterations in the outer loop is minimized.

We will use this result in Chapter 4.

2.3.3 Other Transformations

Matrix-based transformations are relatively new. Before matrix-based transformations were developed, restructuring compilers were constructed using a large number (dozens) of *ad hoc* transformations that were applied individually. (The three transformations of the previous section were all general transformations developed before matrix-based transformation frameworks came into existence.) In this section, we describe several transformations that are not matrix-based, but we find useful in later chapters. Since a major component of our

work deals with loops that are not perfect, we also provide an overview of some of the *ad hoc* transformations that make loop nests perfect.

Loop distribution (LD) as described by Padua [52] is a popular technique for making loops suitable for automatic conversion to vector or parallel form. The technique is also known as loop fission. Although the technique has many uses, a major problem is that loop distribution cannot break a strongly-connected component in a dependence graph. It is this problem that most researchers have attempted to address. In addition, a minor problem is that loop distribution can make one loop nest into a large number of nests. If each of these nests needs a barrier synchronization, then loop distribution can be expensive. (Several compilers use loop fusion to counteract this, but loop fusion is difficult if the loop nests are transformed with unimodular transformations because the compiler may not be able to match the resulting loop bounds.)

There is a simple method for making perfect loop nests from imperfect loop nests by introducing conditional statements. This technique is understood with an example:

```
DO i1 = n1, N1                          DO i1 = n1, N1
   P2[i1]                                  DO i2 = n2, N2
   DO i2 = n2, N2                             DO i3 = n3, N3
      P3[i1, i2]                                 IF i3 = n3 THEN
      DO i3 = n3, N3                                IF i2 = n2 THEN P2[i1]
         S[i1, i2, i3]                              P3[i1, i2]
      END DO                                     END IF
   END DO                                        S[i1, i2, i3]
END DO                                        END DO
                                           END DO
                                        END DO
```

Most researchers cite Abu-Sufah's non-basic-to-basic loop transformation [1] when referring to this technique, but it is probably much older. (Lamport [43] mentions the tech-

nique as early as 1974.) M. E. Wolf [67] points out an important condition for the legality of the non-basic-to-basic loop transformation; one that has frequently been omitted. The condition asserts

for all loop nests j,

$$\mathtt{lb}_j \leq \mathtt{i}_j \leq \mathtt{ub}_j \implies \mathtt{lb}_{j+1}(i_j) \leq \mathtt{ub}_{j+1}(i_j)$$

must be true. Essentially, this states that we cannot move the statements into the body of a loop if there exists the possibility that the loop body will not be executed. This method has the advantage that it is easy to perform and it is legal for a broad range of imperfect loop nests. It has the serious disadvantage that it shifts the decision making from compile-time to run-time and the extra computation takes place in the body of the innermost loop. For some architectures (superscalar, for example) these conditionals can have a significant effect on the execution speed.

M. J. Wolfe briefly discusses imperfect loop nests in [72]. He begins with the non-basic-to-basic approach but points out that the same dependences that prevent loop distribution will frequently prevent loop interchange.[5] Regardless, he recognizes that there is a class of loops for which loop interchange is still legal and presents the (complex) conditions needed to interchange these imperfect loop nests. The legality conditions make this method difficult to perform plus the smaller class of loop nests that are suitable is a drawback.

In his Ph.D. thesis, M. E. Wolf [67] discusses imperfect loop nests. Similarly, he starts with Abu-Sufah's non-basic-to-basic method but he states that under certain conditions, after the unimodular transformations, some of the `if` statements can be moved out of the

[5]Although Wolfe discusses loop skewing, he does not indicate that this problem can be avoided by the combined effects of loop skewing and loop interchange.

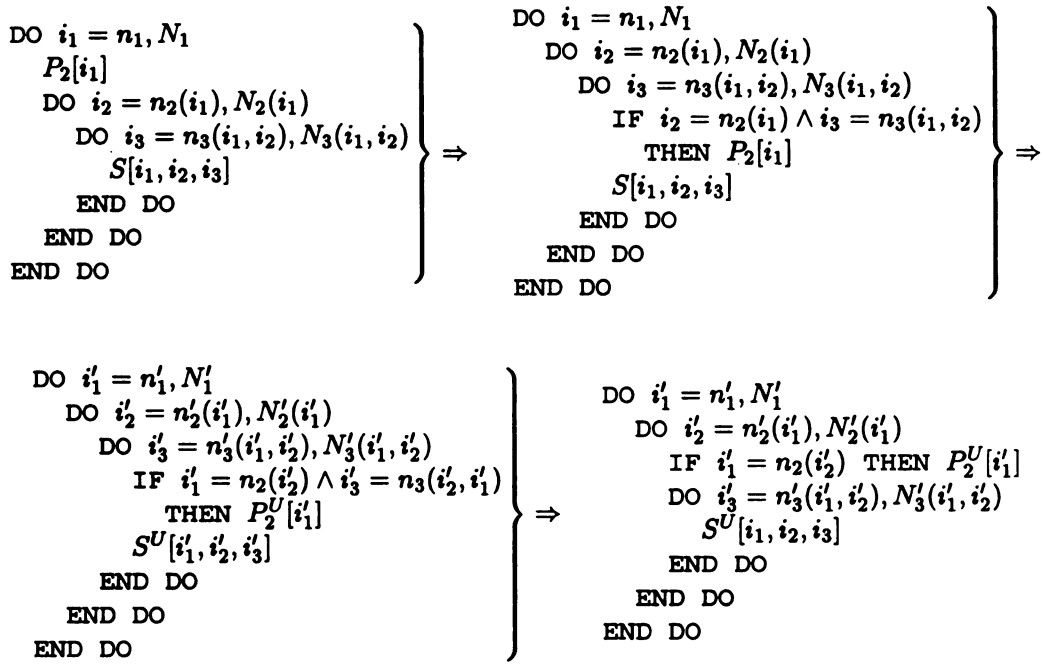

Figure 2.2: four steps in Wolf's approach

innermost loop and part of the condition removed. In Figure 2.2, we have shown the four steps needed to transform a loop this way: shown first is the loop nest before transformation; then it is shown after the non-basic-to-basic transformation; next, it is shown with a unimodular transformation (interchanging the two outer loops); and finally it is shown with the moving of the conditional.

Moving the **if** in this example hinges on the fact that the inner loop is not changed by the transformation. That is, $i'_3 = n_3(i'_2, i'_1)$ is only true at the beginning of the execution of the i'_3 loop. See Wolf [67] for more details.

Wolf claims (p. 51) that "if the unimodular transformation is a skew, then the non-perfectly [*sic*] nested portions would simply be moved in and moved out again [to their

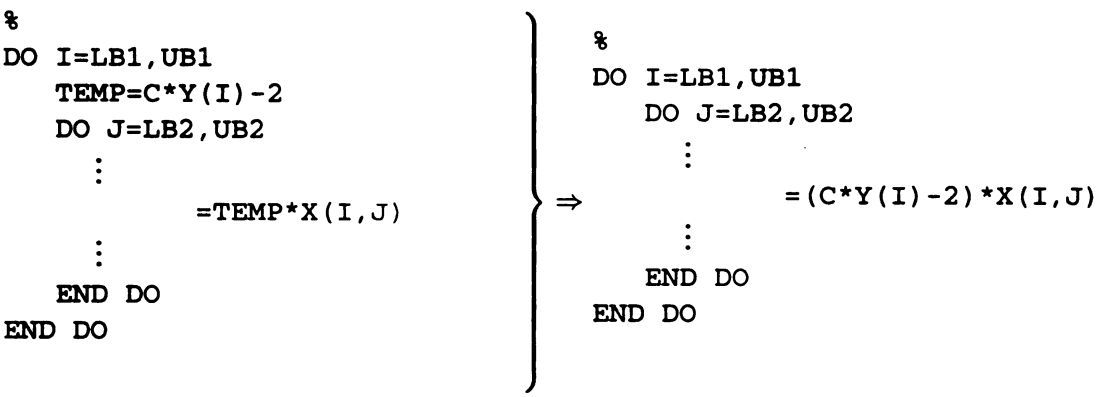

Figure 2.3: SFS method demonstrated

original positions]." This makes performing the transformation (on imperfect loop nests) for loop skewing easy, but it remains complex for general unimodular transformations. Plus, the transformation may only improve upon the non-basic-to-basic transformation in certain cases.

In Chapter 3, we find one of the *ad hoc* techniques called Scalar Forward Substitution (SFS) useful. This transformation has also been called Expression Folding and Scalar Propagation. In certain cases, this technique is useful for making loop nests perfect. SFS will propagate an expression forward within a scope (in our case, a DO loop) when there is exactly one def (that does not involve the previous value of the variable) and one or more subsequent uses. The meaning of def and use are given in the text by Aho, Sethi, and Ullman [2]. The SFS method is demonstrated in the example in Figure 2.3.

SFS is useful in parallel environments because it removes data dependences. As a side-effect, SFS can make an imperfect loop nest perfect. Frequently, SFS will enable a loop to be executed in parallel. SFS actually increases the amount of computation, but the negative

effect of the extra computation is usually much smaller than the positive effect of parallel execution. In fact, SFS is the inverse of common subexpression elimination [2], which is used in many optimizing compilers for scalar machines to remove the extra computation.

To summarize, each of the above transformations handles some imperfect loop nests and each transformation has its own restrictions on when it may be used. No single transformation technique has proven to be clearly superior.

2.3.4 Performance Metrics

Even before restructuring compilers were developed, optimizing compilers have used models and functions as a criteron for performing transformations. Some static models are intuitive or come from the technical data available about the hardware. For example, replacing a branch to an unconditional branch instruction sequence with a single branch is clearly performance-improving. For simple microprocessors, it is easy to determine from the hardware reference when replacing a multiply with multiple additions will yield a performance improvement.

Higher level decisions, such as whether to distribute data or how to order loop transformations or even how many processors to use, cannot be made with simple, static performance models. Many systems have used run-time data to guide compiler decisions. In this subsection, we discuss some of the work related models we have developed for restructuring compilers in Chapter 5.

The use of prior performance data to predict future performance and, ultimately, to improve performance is not novel. Analysis and predication have been used to under-

stand parallel algorithms for some time. For example, Bodin *et al.* [15] extend a modeling method to the BBN GP1000 shared-memory multicomputer that was originally developed by Wijshoff *et al.* [28] for the Alliant FX/8. Both of these studies were interested in revealing the behavior of the memory subsystem. The results were not intended to guide a compiler algorithm.

Mehra *et al.* [50] model the operations of a simulation language BDL for two parallel applications on a iPSC/860 hypercube. They use profiling information to build two analytical models, one for each parallel application. Since these particular applications are re-used often, the prediction tool gives the user the ability to determine in advance how long the program is likely to take and then tune the application to make more effective use of the machine for simulations.

More recently, Fahringer [25] and Fahringer *et al.* [26] used several of the parameters we are using plus cache and specific network information to accurately estimate the performance of loop nests. This information is used to drive a tool that is part of the Vienna Fortran Compilation System (VFCS). The researchers document the fact that the prediction tool can be used to discriminate between different transformations to determine the best one for performance. Their accuracy is derived from the detailed understanding of the communication subsystem and the effects of cache on their multicomputer. Unfortunately, that information varies in NoWs. The VFCS does support NoWs but the NoW version does not include the performance prediction tool.

A. Dierstein *et al.* [22] also uses run-time information to direct compiler transformation decisions. They use their performance metric to successfully perform automatic data distribution and parallelization on an iPSC/860. Similar to the VFCS, their metric depends on

detailed information about communication times in the iPSC/860. In our problem domain, compiling for NoWs, this information is not available because communication subsystems vary and the network's actual performance cannot always be inferred from the reported performance figures.

More recently, K. Kennedy and U. Kremer have introduced a tool for automatic data layout for distributed memory machines [40]. Their approach is to use performance information in conjunction with a search algorithm to present to the programmer, prior to compilation, a set of possible data layouts. Their goals are similar to ours, especially with respect to using prior performance data, but they focus soley on the data layout issue and (apparently) leave the code transformations to the compiler. Others [20] have shown that data and code transformations cannot be considered separately without loss of performance.

Chapter 3

Loop Nests: Taxonomy, Statistics, and Compiler Design

> Errors using inadequate data are much less than those using no data at all.
>
> **Charles Babbage**

A majority of the performance-increasing techniques developed over the last twenty years assume that loop nests are perfect. Researchers, including Kelly and Pugh [37], have begun to question this assumption as new developments — such as the various matrix-based transformations — continue to be limited to perfectly nested loops. This assumption was not a concern with the older techniques because restructuring compilers performed a large number of transformations and some of the *ad hoc* transformations would convert a portion of the imperfect loop nests into perfect loop nests. If restructuring compilers are based on the application of a single transformation — such as a unimodular transformation — then imperfect loop nests are ignored. In this chapter, we analyze how often perfect loop nests

occur in scientific FORTRAN codes, analyze the effects of specific transformations on the codes, and describe a compiler that is fundamentally based on unimodular transformations but is not as restrictive with respect to perfectly nested loops.

A preliminary study suggested that two well-known transformations could be used effectively to make loop nests perfect. In the first half of this chapter, we follow up on this study by coding these two transformations in a FORTRAN compiler and measuring their effectiveness on a large number of scientific FORTRAN programs. The results are presented in 3.1. We measure effectiveness by counting the number of perfectly nested loops before and after the application of the transformations. Our concern here is with the ability to enable other performance-increasing transformations, such as the unimodular transformations, and not with the direct benefits (increased performance) of these transformations.

In the second half of this chapter, we discuss the high-level design of a restructuring compiler that is not restricted to perfect loop nests. We discuss how to integrate the apperfect-to-perfect transformations and unimodular transformations without violating the basic philosophy behind unimodular transformation theory.

3.1 Loop Statistics

In this section we present the results of a static analysis of loop nests. First, we state the definitions used to classify loop nests. Next, our methodology is described. Finally, we present the data gathered and discuss its implications.

Our preliminary investigation indicated SFS and LD were promising techniques to convert imperfect loop nests into perfect loop nests. We hypothesized that combining these two

would be sufficient. We did not include the other transformations because of the previously mentioned disadvantages, the complexity of the transformation, or the inherent limited potential. Our goal is to identify which loops are handled effectively by the composition of SFS and LD and determine the significance of the loops that are not handled.

3.1.1 Definitions

We defined a *perfect* loop nest earlier. In this section we expand upon this definition and build a hierarchy of loop nest classes. Our definitions are pragmatic and therefore, may lack a sense of mathematical elegance. The techniques used to handle loop nests that are not perfect are limited to certain loop structures. We establish a class based on this characteristic. This class is a superset of the perfect loop nests but still does not encompass all loop nests. We call this class of near-perfect loops *apperfect.*[1] (While not all techniques handle all of the loops in this class, we are not aware of any automatic techniques that handle loops outside of this class.) Our definition of an apperfect loop nest is one whose body consists of any combination of SESE sequences of non-loop statements and apperfect loops.

The third class is *general* and includes all legal FORTRAN loops. The examples in Figure 3.1 are general loop nests but not apperfect. The *imperfect* class of loops includes all general nests that are not perfect. This definition of imperfect matches the common usage of the term. Figure 3.2 shows the relationships between these classes.

[1] We have created this word using the prefix *ap-* to suggest that these loops are almost perfect. The definition implies that some transformation exists that will make the loop nest perfect.

```
C This example shows a loop
C nest within the structure of
C a conditional.
 DO I=1,N
  IF ( T(I).EQ.0 ) THEN
   DO J=1,N
    S
   END DO
  END IF
 END DO
```

(a)

```
C This example shows a loop body
C that is not SESE. (There is
C is branch out of the loop body.)
 DO I=1,N
  DO J=1,N
   S
   IF ( ALPHA.LT.ERR ) THEN
    GOTO 10
   END IF
  END DO
 END DO
```

(b)

```
C This example shows a loop body that is not
C SESE because the RETURN is effectively a
C branch out of the loop body.
 DO I=1,N
  DO J=1,N
   S
   IF ( ALPHA.LT.ERR ) THEN
    RETURN
   END IF
  END DO
 END DO
```

(c)

Figure 3.1: three examples of general loop nests that are not apperfect

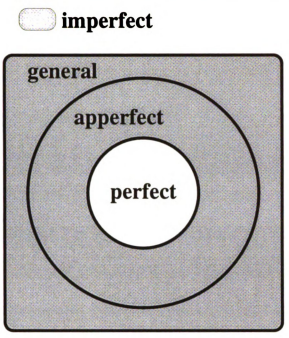

Figure 3.2: relationships between perfect, apperfect, general, and imperfect

In addition to the formal classes described above, we give special consideration to certain loops. The first of these special classes is loop nests that have sequential I/O statements. The execution order of the iteration space for these loops is critical because transforming these loop nests requires a number of assumptions that may be unreasonable. (We assume that the input records are ordered; we also assume that the order of the output records is significant.) Because we are concerned primarily with unimodular transformations and because singly-nested loops cannot be transformed with unimodular transformations, we also separate loops nests of depth one (singly-nested loops).

Methodology. We examined 25,000 lines of scientific FORTRAN code and classified all of the loop nests. In order to process the sample data, we developed a program to parse

FORTRAN, perform the necessary transformations, and classify the loop nests. We used the parser and database library from the Sigma Toolbox 0.2a (which, in turn, was based on the SIGMACS project) [14, 29, 63]. Although the library included subroutine calls to do loop distribution and scalar forward substitution, we rewrote these calls to make them more comprehensive and robust. We relied on the toolkit's data dependence tests, def/use calculations, and unparse routines to complete our source-to-source compiler.

When classifying the loops, our program first checks each loop to see if the loop contains any I/O statements. If so, the loop is classified "Order-Critical." Otherwise, it checks the depth of the loop and singly-nested loops are marked as such. Finally, it tests for perfect, apperfect, and general, in that order. The summarized results are listed package-by-package in Table 3.1.

3.1.2 Discussion

The packages we have selected are well-known FORTRAN codes that are available to the public on the Internet. (We retrieved these via Netlib [23].) Linpack is a collection of linear algebra subroutines. The Misc package includes subroutines that were collected from various sources — for example, samples used in other research papers to demonstrate their techniques. The Nascodes is a collection of five NAS benchmarks. The Svdpack contains a number of singular value computation subroutines. Toeplitz is a package that performs Toeplitz matrix computations (Toeplitz matrices are special cases of persymmetric matrices). In all cases, if multiple versions of the routines existed (for example, single precision version, double precision version, complex version, *etc.*), we chose only to include one ver-

Table 3.1: loop statistics with various transformations applied

No Transformations

Application	#Nests	#Perfect	#Apperfect	#General	#Order-Critical	#Singly-nested
linpack.f	146	2	8	3	0	133
misc.f	153	13	24	2	11	103
nascodes.f	82	14	15	2	3	48
svdpack.f	151	18	17	21	8	87
toeplitz.f	49	7	5	4	0	33
TOTAL	581	**54**	**69**	**32**	22	404

Loop Distribution

Application	#Nests	#Perfect	#Apperfect	#General	#Order-Critical	#Singly-nested
linpack.f	150	4	6	3	0	137
misc.f	179	23	14	2	11	129
nascodes.f	91	24	13	2	3	49
svdpack.f	154	19	16	21	8	90
toeplitz.f	60	11	3	4	0	42
TOTAL	634	**81**	**52**	**32**	22	447

Scalar Forward Substitution

Application	#Nests	#Perfect	#Apperfect	#General	#Order-Critical	#Singly-nested
linpack.f	146	4	6	3	0	133
misc.f	153	19	18	2	11	103
nascodes.f	82	22	7	2	3	48
svdpack.f	151	18	17	21	8	87
toeplitz.f	49	7	5	4	0	33
TOTAL	581	**70**	**53**	**32**	22	404

Scalar Forward Substitution followed by Loop Distribution

Application	#Nests	#Perfect	#Apperfect	#General	#Order-Critical	#Singly-nested
linpack.f	152	7	3	3	0	139
misc.f	183	31	6	2	11	133
nascodes.f	95	34	3	2	3	53
svdpack.f	155	19	16	21	8	91
toeplitz.f	56	11	3	4	0	38
TOTAL	641	**102**	**31**	**32**	22	454

sion. Normally, the double precision version was chosen. Our intent is to keep the original relative proportions. In addition, if a package uses the same subroutines for different algorithms, we only include the subroutine once. The programs TSVD1 and TSVD2 in Svdpack both call DGEMM. Since TSVD1 and TSVD2 come in separate, stand-alone FORTRAN files, DGEMM appears twice. To avoid skewing our results, we applied the following rule: if two subroutines have the same name and are textually (line-by-line) identical, we removed one of the routines from our sample. If the two routines performed the same function but were slightly different (perhaps written by different authors), we left both subroutines intact. All of the subroutines included in this study are listed in Table 3.2.

There is a practical consideration to take into account regarding these results. First, we are using Sigma's data dependence tests. These routines are fast but not exact. We have observed that the Parafrase-2 compiler distributes a loop nest when ours does not because Parafrase-2's extensive data dependence tests indicate that LD is legal. Since we cannot safely distribute the loop nest based on Sigma's data dependence test, we have to leave the loop nest apperfect. This means our analysis may err by being overly conservative.

It is important to note that the general class does not change under any of these transformations. This is to be expected since these loops are distinguished by the fact that we do not have an automatic loop transformation to handle them.

An encouraging result is that the two transformations (SFS and LD) are fairly effective. More encouraging is that together they are significantly better than either one alone. The data does not validate our hypothesis at the beginning of this section — SFS and LD are not sufficient — but it does suggest we have made progress. In fact, we can observe that a large number of the troublesome apperfect loops (16 of 31) are from a single package, Svdpack.

Table 3.2: listing of subroutines included in each package

misc.f	
1000d.f:	linear equation solver
ddasrt.f:	differential/algerbraic eq. of the form $F(T, Y, Y') = 0$
na1.f:	solution to sys of linear equations (avoiding near breakdown)
na2.f:	generate B-nets of any box splines on a 3D mesh
ode.f:	integrates a sys of n first order differential equations of the form $\frac{dy(i)}{dt} = f(t, y(1), y(2), ..., y(n))$
wolfe.f:	example from [72]

nascodes.f	
buk.f:	bucket sort
cgm.f:	conjugate gradient method
embar.f:	embarassingly parallel
fftpde.f:	3D FFT PDE
mgrid.f:	simple multigrid solver (3D potential field)

svdpack.f	
las2.f:	Single vector Lanczos for $A'A$ eigensystems
bls1.f:	Block Lanczos for equivalent 2-cyclic eigensystems
bls2.f:	Block Lanczos for $A'A$ eigensystems
sis1.f:	Subspace iteration (Rutishauser's ritzit) for equiv. 2-cyclic eigensystems
sis2.f:	Subspace iteration (Rutishauser's ritzit) for $A'A$ eigensystems
tms1.f:	Trace minimization for shifted 2-cyclic eigensystems using Ritz-shifting
tms2.f:	Trace minimization for shifted $A'A$ eigensystems using Ritz-shifting and Chebyshev polynomial acceleration
blas.f:	Level 1, 2, and 3 Basic Linear Algebra Subroutines

linpack.f	
dchdc.f:	Cholesky decomposition
dchdd.f:	augmented Cholesky decomposition
dchex.f:	updates the Cholesky factorization
dchud.f:	updates an augmented Cholesky decomposition
dgbco.f:	factors a real/complex band matrix by Gaussian elimination
dgbdi.f:	computes the determinant of a band matrix
dgbfa.f:	factors a band matrix by elimination
dgbsl.f:	solves the real/complex band system $Ax = b$ or $A^{\mathrm{T}}x = b$
dgeco.f:	factors a matrix by Gaussian elimination

Table 3.2: listing of subroutines included in each package (continued)

linpack.f (continued)	
dgedi.f:	computes the determinant and inverse of a matrix
dgefa.f:	factors a real/complex matrix by Gaussian elimination
dgesl.f:	solves the real/complex system $Ax = b$ or $A^{\mathrm{T}}x = b$
dgtsl.f:	solves a general tridiagonal matrix given a rhs
dpbco.f:	factors a real/complex symmetric pos. definite band matrix
dpbdi.f:	computes determinant of symmetric pos. definite band matrix
dpbfa.f:	factors a symmetric positive definite band matrix
dpbsl.f:	solves the symmetric positive definite band system $Ax = b$
dpoco.f:	factors a symmetric positive definite matrix
dpodi.f:	computes determinant and inverse of symmetric pos def matrix
dpofa.f:	factors a symmetric positive definite matrix
dposl.f:	solves the real/cmplx symmetric pos definite system $Ax = b$
dppco.f:	factors a real/cmplx symmetric pos definite packed matrix
dppdi.f:	computes det and inverse of a symmetric pos definite matrix
dppfa.f:	factors a real/cmplx symmetric pos definite packed matrix
dppsl.f:	solves the real/cmplx symmetric pos definite system $Ax = b$
dptsl.f:	given a pos definite tridiagonal matrix and a rhs will find solution
dqrdc.f:	Householder transformations to compute QR factorization
dqrsl.f:	computes coord. transform., project., & least squares solutions
dsico.f:	factors symmetric/symmetric/Hermitian matrix by elimination
dsidi.f:	computes det, inertia and inverse of a sym/sym/Herm matrix
dsifa.f:	factors a symmetric/symmetric/Hermitian matrix by elimination
dsisl.f:	solves the symmetric/symmetric/Hermitian system $Ax = b$
dspco.f:	factors a symmetric/symmetric/Hermitian packed matrix
dspdi.f:	computes det, inertia and inverse of a sym/sym/Herm matrix
dspfa.f:	factors a symmetric/symmetric/Hermitian packed matrix
dspsl.f:	solves the symmetric/symmetric/Hermitian system $Ax = b$
dsvdc.f:	reduces matrix to diagonal by orthogonal/unitary transform.
dtrco.f:	estimates the condition number of a triangular matrix
dtrdi.f:	computes the determinant and inverse of a triangular matrix
dtrsl.f:	solves triangular systems of the form $Tx = b$ or $T^{\mathrm{T}}x = b$

toeplitz.f	
tsld.f:	Toeplitz (real) (user interface to TSLD1)
tsld1.f:	Toeplitz (real)
tslz.f:	Toeplitz (complex) (user interface to TSLZ1)
tslz1.f:	Toeplitz (complex)
cslz.f:	Circulant (complex)
cqrd.f:	Column-circulant (real orthogonal factorization)
cqrz.f:	Column-circulant (cmplx orthogonal factoriz.)
tgsld.f:	Block-Toeplitz (real general blocks)
	(interface to tgsld1)

Table 3.2: listing of subroutines included in each package (continued)

toeplitz.f (continued)	
tgsld1.f:	Block-Toeplitz (real general blocks)
tgslz.f:	Block-Toeplitz (complex general blocks) (user interface to tgslz1)
tgslz1.f:	Block-Toeplitz (complex general blocks)
ctslz.f:	Block-circulant (complex Toeplitz blocks)
ccslz.f:	Block-circulant (complex circulant blocks)
cgslz.f:	Block-circulant (complex general blocks)
salwz.f:	Block-circulant (complex service routine for computing direct or inverse discrete Fourier transformations)
ctgslz.f:	3-level block-circulant (TG blocks)
cctslz.f:	3-level block-circulant (CT blocks)
cccslz.f:	3-level block-circulant (CC blocks)
ccgslz.f:	3-level block-circulant (CG blocks)

This suggests that many users will find SFS and LD useful but selected users who require a package such as Svdpack, may observe poor performance. If the Svdpack package is ignored, we see that of the 109 relevant loops (perfect, apperfect, and general) shown in Table 3.1, three-quarters are perfect after both transformations are applied. When Svdpack is included, 31 loop nests remain apperfect. This surprising result led us to investigate further the application and its loops.

We extended our compiler to write the apperfect loops into a separate file after both transformations were applied. Next, we inspected the loops manually. We noticed a number of similarities and decided to further classify these loops. We searched for characteristics that would cause both transformations to fail. We identified four characteristics of loop bodies: (I) presence of a recurrence statement; (II) presence of an induction variable; (III) multiple DO loops; and (IV) the presence of very complicated data dependences. We have

Table 3.3: loop characteristics that cause SFS and LD to fail and their frequency

Recurrence:	24
Induction Variables:	8
Multiple DO's:	8
Complex Data Dependences:	2

included samples of the four cases in Figure 3.3. The number of times each of these cases occurred is shown in Table 3.3. Note that the total exceeds 31 because some nests exhibited more than one of the characteristics.

These characteristics are not new discoveries. Recurrence statements are discussed by Eigenmann, *et al.* [24] and Allen and Kennedy [3]. These statements tend to be difficult to parallelize. Usually, the most effective technique is to replace the statement with a machine-optimized library call. In essence, change the algorithm. In [24] the authors referenced a study by Meier and Eigenmann [51] that showed this library-call solution yielded a 50% increase in performance for a Conjugate Gradient algorithm on the Cedar architecture. In [3], Allen and Kennedy include an example that has an induction variable and mention a transformation "induction variable substitution" (described by M. J. Wolfe [73]). This transformation removes the induction variable by computing its value from the normal loop index and loop constants. (Induction variables are actually a special case of recurrence statements.) When multiple DO loops occur, as in Figure 3.3(c), it is either

```
do j = 1, ip
  acc = 0.0d0
  do i = 1, ip
    if (iflag .eq.  0) then
      acc = acc+a(j,i)*
        b(ibstar+i)
    else
      acc = acc+a(i,j)*
        b(ibstar+i)
    end if
  end do
  c(icstar+j) = acc
end do
```

(a)

```
do j = 1, n
  temp = zero
  do i = 1, m
    temp = temp+a(i, j)*x(ix)
    ix = ix+incx
  end do
  y(jy) = y(jy)+alpha*temp
  jy = jy+incy
end do
```

(b)

```
do kb = 1, nm1
  do i = n-kb+1, n
    work(i) = a(i, n-kb)
    a(i, n-kb) = 0.0d0
  end do
  do i = n-kb+1, n
    t = work(j)
    call daxpy(n,t,a(1,j),
      1,a(1,n-kb),1)
  end do
end do
```

(c)

```
do i = k+1, n
  a(i, k) = a(i-1, k+1)*
    a(k+1, k+1)
  do j = k+1, n
    a(i, j) = a(i, j)+
      a(i, k)*a(k, j)
  end do
end do
```

(d)

Figure 3.3: These loops come from our sample of FORTRAN codes. The nest **(a)** has a recurrence statement, the nest **(b)** has an induction variable (and a recurrence statement), the nest **(c)** has multiple DO's, and the dependences in the nest **(d)** made it impossible to apply any of our transformations.

because the loops are in the same strongly connected component or the data dependence tests could not establish that the loops belong in separate strongly connected components. Either a change in the algorithm is required or a more accurate dependence test is needed. Likewise, cases with the characteristic of very complicated data dependences may only benefit from an algorithmic change. In general, these problems are beyond any of the loop transformations commonly available. An interactive tool may be important for these situations: analysis and transformations could be handled in bulk, automatically, but the difficult cases — ones that require a different algorithm — could be highlighted by the tool.

We have found that of the remaining 31 apperfect loop nests, most require changes to the algorithm. Additional apperfect-to-perfect transformations have little potential for improvement over SFS and LD. Thus, we conclude that most of the loops in the apperfect class no longer represent a challenge to researchers.

3.2 Compiler Design

The data in 3.1 is important because it allows us to design a restructuring compiler that is based on unimodular transformations (UTs) but is not limited to perfect loop nests. In this section, we discuss the differences between restructuring compilers that are based on UTs and others based on generate-and-test. We show how to integrate scalar forward substitution and loop distribution with UTs and argue that the resulting compiler is still essentially a UT-based compiler.

Generate-and-Test versus Unimodular Transformation Theory. Before the emergence of unimodular transformation theory, restructuring compilers either allowed the user to select the order in which to apply the loop transformations or the compiler had to generate its own order. As researchers advanced the state of restructuring compilers, it was realized that a single, static ordering decided *a priori* will not be good in all cases — certainly not optimal. Whitfield and Soffa [66] discuss this in terms of transformations enabling (or disabling) other transformations. In practice, over a large collection of loops, one might observe that transformation X will frequently enable transformation Y and at the same time observe that transformation Y will sometimes enable transformation X. Thus, we should not give a fixed order to these two transformations. M. J. Wolfe [70] shows examples involving scalar and parallel transformations where a particular ordering is good in one case and bad for another. Likewise, Wolfe gives two examples where the same is true for the reverse ordering. Thus, a restructuring compiler that desires optimal or near optimal transformations needs to follow a "brute force" algorithm: generate an order, test for legality and optimality, generate another ordering, test, and so on. This approach is necessary because of the large number of transformations and because the transformations' *ad hoc* and special nature make the interactions too complex to manage.

In contrast, compilers with algorithms based on unimodular transformation theory avoid the brute force approach. By limiting the transformations to just three elementary transformations, the theory disencumbers the complex interactions. Thus, a complex unimodular transformation composed of many elementary transformations becomes manageable. Specifically, we can determine the legality of the transformation, the new loop bounds, and the new loop body with ease. And most important, one can choose very precise goals and

design algorithms to find the desired unimodular transformation that maximizes that particular goal in polynomial time. This is a substantial development over compilers based on generate-and-test.

Each approach has its advantages and disadvantages. There are a large number of transformations available to a generate-and-test compiler. Some of these are crucial for converting apperfect loop nests to perfect loop nests. But with the generate-and-test compilers, we only have the brute force approach to ordering the transformations. Compilers based on unimodular transformations have polynomial time algorithms that guarantee optimality for particular goals, but unimodular transformations can only be applied to perfect loop nests.

Thus, if we can join the apperfect-to-perfect loop transformations discussed in the previous section with unimodular transformations then we have removed the disadvantage to compilers based on unimodular transformation theory. We indicate how this can be done in the next section.

Integrating SFS/LD with Unimodular Transformations. There are two options to consider when joining new transformations to the unimodular transformation theory. The first option is to make these transformations part of the elementary transformations and reestablish the proofs and algorithms that are principal to the theory. The second option is to determine if there is a fixed order that can be applied to the transformations to assure optimality. Although it may be possible to incorporate these transformations into the theory, it would significantly modify the framework. Also, the changes to the proofs would not be trivial. Instead, we show that SFS followed by LD is never worse than any other ordering sequence.

To establish this ordering, we consider the four possibilities: no transformations, SFS only, LD only, and LD followed by SFS. We will show that SFS followed by LD is always as successful as any other ordering. We define successful to mean that the transformed loop nests: (I) are perfect, if possible; (II) result in the fewest number of nests (*i.e.*, fewest barrier synchronizations); and (III) contain the minimum extra computation. We assume that (I) is the most important because it means that unimodular transformations will succeed at parallelizing the loop and that (III) is the least important.

No Transformation. SFS is always as good as or better than no transformations because we are able to remove the bad effects of SFS (extra computation) if needed. If we find that after applying SFS that the unimodular transformation is unable to parallelize the loop, then common subexpression elimination and strength reduction (both are well-known transformations [2]) will undo SFS. So either SFS contributes to the parallelization or it is equivalent to not applying any transformations.

Similarly, LD is always as good as or better than no transformations. If LD has been applied and does not contribute to the parallel execution of the DO loop, it is easy to undo LD with loop fusion. (If the loop is parallelized by a unimodular transformation, then it is unlikely loop fusion will succeed because of the new loop bounds.) This, too, is common practice as noted by Sarkar [57].

Therefore, if SFS followed by LD is applied to a loop nest, the composition of these two will either contribute to the parallel execution of the loop or we can undo the effects. Thus, it is never worse than no transformations.

SFS Alone. As in the previous argument, the effects of LD can be removed by loop fusion if LD does not contribute to the parallel execution of the loop nest. Thus, SFS followed by LD is for all cases no worse than SFS alone.

LD Alone. Loop distribution uses data dependences between statements to build a dependence graph. The strongly connected components of this graph become separate loop nests. If we are relying on loop distribution to make loop nests perfect, then we desire the graph to be as "weakly" connected as possible. If SFS is legal, then it will always remove at least one dependence (between the "def" statement and one or more subsequent "use" statements). Removing a dependence will never increase the strength of a dependence graph. Thus, performing SFS followed by LD is no worse than performing LD by itself.

LD followed by SFS. Suppose that there exists an example E, where LD followed by SFS is better than SFS followed by LD. Then both transformations must contribute because we have shown that one or the other alone could not have out-performed SFS followed by LD. This condition in conjunction with the fact that SFS cannot disable LD (the previous argument), implies that LD must have enabled SFS. Since LD neither creates nor destroys statements, it must have separated some statements into different loops such that it is now legal to apply SFS. Such separation is impossible because all the occurrences of the scalar being substituted that would make SFS illegal must appear in the same strongly connected component. Therefore, LD could not have separated them. Consequently, example E cannot exist.

Since our intent is to use SFS and LD to make loop nests suitable unimodular transformations, clearly these two come before the unimodular transformation is applied. Thus, we have argued that SFS followed by LD is an optimal ordering. We include loop fu-

sion, common subexpression elimination (CSE), and strength reduction to ensure that our transformations are successful for all cases. This order is shown in Figure 3.4.

3.3 Summary

This chapter has three significant results, presented here and elsewhere [58]. First, analysis of our data has led us to formulate three classes of loop nests (perfect, apperfect, and general) whereas others have only considered two: perfect and imperfect. Although using three classes instead of two is a simple change, it is important because it allows us to identify the relative significance of the loop nests that current restructuring compilers handle well and the loop nests that they do not. A second result is that a significant number of apperfect loop nests can be transformed into perfect loop nests by just two existing transformations. The third result is that loops in our general class, which includes loops for which no automatic general-to-perfect transformation is known, occur frequently in scientific codes. Therefore, we have identified a class of loops that deserves more attention from researchers and suggests that less work on apperfect-to-perfect transformations is needed.

Presented with these results, we draw three conclusions. First, while unimodular transformations assume that loop nests are perfect, this is not a practical limitation for a compiler based on unimodular transformation theory. With the use of scalar forward substitution and loop distribution, we have shown that compilers based on unimodular transformations are no more limited than any other proposed framework.

Second, we found that new apperfect-to-perfect loop transformations have very little potential unless they specifically address the problems shown in Table 3.3. After apply-

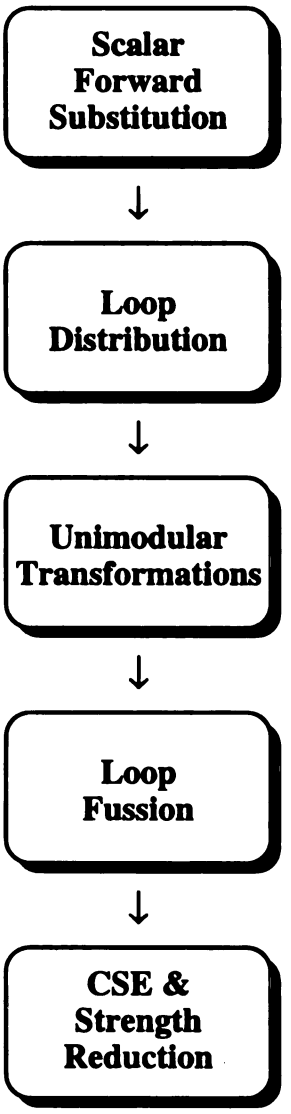

Figure 3.4: optimal ordering of transformations recommended in this chapter

ing scalar forward substitution and loop distribution, most apperfect loops become perfect loops. Since the remaining loops require an algorithmic change, we believe that it is unlikely that these loops will be handled by a loop transformation. They are significant, though, and we believe they do require some form of transformation.

By distinguishing the apperfect loops from the general loops, we established our third result. There are a significant number of loops outside of the apperfect class for which there are no loop transformations available. We suspect that a certain number of these loops have potential and thus deserve more attention from the research community.

Chapter 4

Phase Method

The main finding presented in this chapter is a new algorithm called the *phase method.* It is closely allied to other unimodular transformation algorithms such as Banerjee's [9] except for the important distinction that it accepts imperfectly nested loops.

The result presented is significant for two reasons. Although other techniques exist to convert imperfect loop nests into perfect loop nests — loop distribution as detailed by Padua, Kuck, and Lawrie [52] being the most notable — they cannot always be applied or it may be undesirable to apply them. (Loop distribution, for example, cannot break strongly connected components in a data dependence graph.) The phase method can parallelize code that previous techniques cannot.

Secondly, some researchers are proposing alternative frameworks because unimodular transformations are limited to perfectly nested loops (see Kelly and Pugh [37]). Although this algorithm only handles loop nests of depth two, it strengthens the argument that unimodular transformations are amenable to generally nested loops. This is also significant because others, including Banerjee [11], have expressed interest in solving this problem.

This chapter has two main sections. In 4.1, we show how we generate the data dependence vectors and describe our algorithm. The algorithm is demonstrated with an example in 4.2. We summarize the results of this chapter in 4.3.

4.1 Algorithm

In this section we describe the phase method. First, we calculate the dependence distance vectors for imperfect loop nests such that they include our enlarged iteration space. The next step is to invoke Banerjee's algorithm in Figure 4.1 to produce a unimodular transformation matrix and new loop bounds. Using these outputs we characterize the loop nest and generate code. The most important feature of this approach is that by characterizing the loop nest, our algorithm determines at compile time when to execute the leading statements and trailing statements.

Calculating data dependence vectors has been studied extensively (see the references by Pugh [56]) and no algorithm has emerged as clearly the "best." Thus, in our initial efforts we tried to distance ourselves from any single algorithm. To accomplish this goal, we gave our source program to the dependence analyzer and then adjusted its output to fit our iteration space. Although this was usually effective for doubly nested loops, we have since adopted the method used by M. E. Wolf [67], which is more robust and easily extended to n dimensions.

At the heart of most dependence analyzers, there is an integer constraint solver. We can extract the dependence information in the form we want directly by augmenting the input to the constraint solver within the dependence analyzer. For doubly nested loops, the

INPUT: set of dependence distance vectors and direction vectors and the original loops' bounds

OUTPUT: a matrix U such that the transformed inner loop may be executed in parallel and the transformed outer loop has the minimum number of iterations; specifically these variables are set:

U	a unimodular matrix that transforms the distance vectors and leading, body, and trailing statements
m_1, M_1	the lower and upper bounds of the transformed outer loop
$m_2()$, $M_2()$	functions for the lower and upper bounds of the transformed inner loop

SUMMARY: Banerjee's Algorithm works by considering a list of possible transformations, removing the illegal ones, adding a specific skewing transformation, and then choosing the best remaining choice from the list.

Figure 4.1: An algorithm for finding a loop transformation for fine-grain parallelism is summarized in this figure. The detailed algorithm is labeled "Algorithm 6.2" in [9].

additional constraints are: P_2 executes only when $i_2 = n_2(i_1)$ and E_2 executes only when $i_2 = N_2(i_1)$. (As Wolf notes [67], this is conceptually the same as Abu-Sufah's non-basic-to-basic transformation.) For n-nested loops, the constraints for the leading statements ($2 \leq q \leq p$) would be: P_q executes only when

$$i_{q+1} = n_{q+1}(i_q) \wedge i_{q+1} = n_{q+2}(i_q, i_{q+1}) \wedge \cdots \wedge i_p = n_p(i_q, i_{q+1}, \cdots, i_p)$$

and constraints for the trailing statements would be similar. The effect of augmenting the input to the constraint solver is that all of our dependence vectors are of length p. Otherwise, a dependence involving P_q, for example, would only be of length q (where $q \leq p$).

Once we have the distance and direction vectors, we can invoke the algorithm summarized in Figure 4.1. Although the algorithm was meant for perfectly nested loops, the dependence distance set that we are passing includes dependencies involving the leading and trailing statements. Our only remaining task is to generate the code, based on the resulting matrix and new loop bounds, with the understanding that each index point in the iteration space is not necessarily the same sequence of statements. (It may be P_2, E_2, or S.) We generate the code by characterizing the iteration space. For doubly-nested loops, the characterization is represented with a single variable.

The key to the phase method is the recognition that the "wave" (the outer loop of the transformed code) travels through the iteration space in phases. For a doubly-nested loop, there are three phases. The first phase may be vacuous or both leading and body statements are executed. The middle phase has three exclusive possibilities: only body statements

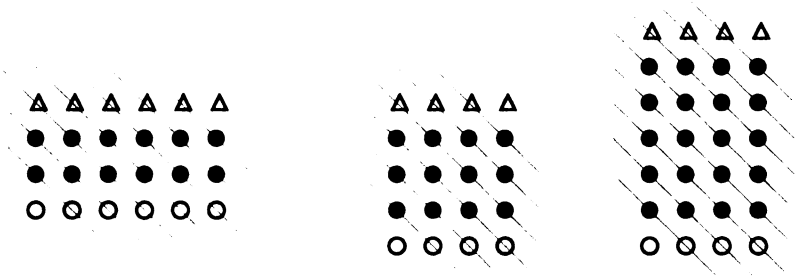

Figure 4.2: There are three simple cases that arise, based on the shape of the dependence graph. The first occurs when the middle phase hits both the leading statements and the trailing statements. The second occurs when the middle case is vacuous. The last case is when we have one or more waves that have neither leading nor trailing statements. (The ○ represents P_2, the • represents S, and the △ represents E_2.)

are executed; leading, trailing, and body statements are executed; or it is vacuous. The third phase, similar to the first, may be vacuous or both trailing and body statements are executed. The characterization of the phases comes from the shape of the iteration space. Consider the same wave on the three different iteration spaces in Figure 4.2: for each case, the middle phase is different. Since the angle of the wave depends on the amount of skewing, it may be the case that in a given phase there may be more waves than the leading or trailing statements. That is, these examples do not illustrate that the leading statements may only be executed once every n waves during the first two phases of execution.

We characterize the iteration space by looking at the middle phase. We use *share* to indicate the shape of the iteration space. When *share* is negative, it indicates that the leading, trailing, and body statements are part of the middle phase. When *share* is positive, the middle phase has only body statements. In all cases $|share|$ is the number of waves in the middle phase. To determine the value of *share*, we must first determine the last wave

to execute a leading statement (A_2) and the total number of waves (w_{tot}). Therefore, we calculate *share* with the following expressions:

A_2	$=\|u_{12}\|(N_1 - n_1)$	number of leading waves that may execute P_2
Ω_2	$=A_2$	number of trailing waves that may execute E_2
w_{tot}	$=\|u_{11}\|(N_1-n_1)+\|u_{12}\|(N_2-n_2)+1$	total number of waves
share	$=w_{tot} - 2\|u_{12}\|(N_1 - n_1)$	characterizes the middle phase
μ	$=u_{12}$	this is the slope of the wave (or μ the skew)

Once we know how many leading statements there are in the middle phase, we can calculate the remaining waves in the first and third phases.

α_2	$=\min\{A_2, A_2 + share\}$	number of waves in the first phase
ω_2	$=\min\{\Omega_2, \Omega_2 + share\}$	number of waves in the last phase

At this point we are ready to generate code. Code generation is simply a matter of generating code for each of the three phases. There are several alternatives to choose from when generating the phases; we chose the simplest one here. The first phase is shown in Figure 4.3(a). When generating code for the middle phase, *share* determines the code. If *share* is zero, then nothing is generated. If $share < 0$ then the code of Figure 4.3(c) is generated. If $share > 0$ then the code of Figure 4.3(d) is generated. The last phase is shown in Figure 4.3(b).

We have presented this algorithm in its simplest form to make the concepts clear. There are variations that we would make in an implementation. The conditional statements in Figure 4.3(a,b,c) can be avoided by replicating the inner loop μ times and changing the step size of the outer loop. Each instantiation of the inner loop would use $W, W + 1, ..., W + \mu - 1$. The step size would be μ. This is feasible because μ is known at compile-time and tends to be small.

```
P_2[n_1, n_2 - 1]
DO W = m_1, m_1 + α_2 - 1
  PAR DO X = m_2(W), M_2(W)
    S^U[W, X]
  END
  IF MOD(W+1, μ) = μ-1 THEN
    P_2^U[W + 1, m_2(W + 1) - 1]
  END
END DO
```
(a)

```
DO W = m_1 + α_2 + |share|, M_1
  IF MOD(M_1 - W, μ) = μ - 1 THEN
    E_2^U[W, M_2(W) + 1]
  END
  PAR DO X = m_2(W), M_2(W)
    S^U[W, X]
  END
END DO
E_2^U[N_1, N_2 + 1]
```
(b)

```
DO W = m_1 + α_2, m_1 + α_2 + |share| - 1
  IF MOD(M_1 - W, μ) = μ - 1 THEN
    E_2^U[W, M_2(W) + 1]
  END
  PAR DO X = m_2(W), M_2(W)
    S^U[W, X]
  END
  IF MOD(W+1, μ) = μ-1 THEN
    P_2^U[W + 1, m_2(W + 1) - 1]
  END
END DO
```
(c)

```
DO W = m_1 + α_2, m_1 + α_2 + |share| - 1
  PAR DO X = m_2(W), M_2(W)
    S^U[W, X]
  END
END DO
```
(d)

Figure 4.3: For doubly nested loops, there are three phases of output during code generation. The first and third phases are shown in **(a)** and **(b)**, respectively. The middle phase depends on the variable *share*. If $share < 0$ then **(c)** is output. If $share > 0$ then **(d)** is output.

Another improvement can be made when the original loop does not have any trailing statements. In this case, the "wave" only travels through two distinct phases (the second and third phases may be merged). If the leading statements are missing, the first and second phases are merged. If both the leading and trailing statements are missing, then all three phases are merged and we get code identical to Banerjee's original algorithm.

4.2 Example

In this section we illustrate the algorithm with an example. The original code is listed with line numbers in Figure 4.4 along with the dependence information. (We used a well-known program, `Tiny`, by M.J. Wolfe to generate the distance vectors [71] and then used the constraints described in the previous section to arrive at these distance vectors.)

Passing the vector set $\{(0,1),(3,0)\}$ and the loop bounds to Banerjee's algorithm [9] results in the following transformation matrix U and new loop bounds $m_1..M_1$ and $m_2()..M_2()$:

$$U = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \qquad \begin{aligned} m_1..M_1 &= 4..20 \\ m_2(x)..M_2(x) &= \lceil \max\{3, x-10\} \rceil .. \lfloor \min\{10, x-1\} \rfloor \end{aligned}$$

Next we calculate:

$$A_2 = 1(10-3) = 7 \qquad \Omega_2 = 7$$

$$w_{\text{tot}} = 1(10-3)1(10-1)+1 = 17 \qquad share = 17 - 2(1)(10-3) = 3$$

$$\alpha_2 = \min\{7,10\} = 7 \qquad \omega_2 = \min\{7,10\} = 7$$

Now, using the templates from Figure 4.3(a,c,d), we get the code in Figure 4.5.

code

```
 5 | DO I = 3,10
 6 |   C(I) = 1/(A(I-3,1)*A(I-3,1)-1)
 7 |   COLSUM(I) = 0
 8 |   DO J = 1,10
 9 |     A(I,J) = C(I)*X(I,J)
10 |     COLSUM(I) = COLSUM(I)+A(I,J)
 8 |   END DO
 5 | END DO
```

Dependence Information

flow	6:C(I) → 9:C(I)	(0,1)
flow	7:COLSUM(I) → 9:COLSUM(I)	(0,1)
outp	10:COLSUM(I) → 7:COLSUM(I)	(0,1)
flow	9:A(I,J) → 6:A(I-3,J)	(3,0)
flow	10:COLSUM(I) → 10:COLSUM(I)	(0,1)

Figure 4.4: example code for phase method

```
C     Enter phase 1 ...
      C(3) = 1/(A(0,1)*A(0,1)-1)
      COLSUM(3)=0
      DO W=4,10
        PAR DO X=MAX(3,W-10),MIN(10,W-1)
          A(X,W-X)=C(X)*X(X,W-X)
          COLSUM(X)=COLSUM(X)+A(X,W-X)
        END DO
        T1=MAX(3,W-9)-1
        C(T1)=1/(A(T1-3,1)*A(T1-3,1)-1)
      END DO
C     Enter phase 2 ...
      DO W=11,13
        PAR DO X=MAX(3,W-10),MIN(10,W-1)
          A(X,W-X)=C(X)*X(X,W-X)
          COLSUM(X)=COLSUM(X)+A(X,W-X)
        END DO
      END DO
C     Enter phase 3 ...
      DO W=14,20
        PAR DO X=MAX(3,W-10),MIN(10,W-1)
          A(X,W-X)=C(X)*X(X,W-X)
          COLSUM(X)=COLSUM(X)+A(X,W-X)
        END DO
      END DO
```

Figure 4.5: resulting code

Notice in the code, when the skew (μ) is one or zero, the condition is not needed (it is always true). We introduce `T1` and reduce constant expressions to simple values for readability.

4.3 Summary

In this chapter, which appreared previously in a shortened form [59], we argue that unimodular transformations on loop nests are not fundamentally limited to perfectly nested loops. We extend the notation of an iteration space to include imperfectly nested loops and present a unimodular transformation algorithm that generates parallel code for imperfectly nested loops of depth two. Furthermore, this algorithm automatically produces parallel code in cases where previous work on automatic imperfect loop transformations could not.

Chapter 5

Performance Metric for NoWs

The Shape of Things to Come
H.G. Wells [65]

Typically, researchers use the speedup curve or performance curve[1] to show the effectiveness of a parallel solution. The execution time for several runs of an application are graphed against the number of processors while keeping the size of the problem fixed.

Although the general shape is well-recognized and its cause well-understood, it is not simple to identify the critical characteristics of the performance curve of an application prior to its execution. The precise shape of the curve depends on many aspects of the system but the primary factors are the amount of parallel computation, communication required by the application, and the hardware characteristics of the NoW. A typical speedup curve is shown in Figure 5.1 with the number of workstations on the abscissa and the execution time

[1] Sometimes the speedup curve is shown with execution time inverted since performance is the reciprocal of execution time. Throughout the chapter we refer to the components of the execution time, so it is natural to not invert the execution time. In this chapter, we continue to call it a performance curve.

on the ordinate axis. The overall execution time is marked with a solid line at the top of the bar. The time spent doing parallel computation and interprocess communication are shown as the dark and light shaded areas, respectively. The gap between the two is overhead that is not measured as communication or computation by our system but does contribute to the overall execution time of the application. By developing a metric that models these main components and the overall execution time, a restructuring compiler framework can make better transformation decisions. Identifying the cost of communication is especially important, as evidenced by the large number techniques developed to address this issue [30, 32, 18, 46, 35].

To develop a metric, we need to understand the performance curve in terms of a variable number of processors and a variable problem size. Whereas the number of workstations is a simple parameter to identify, the problem size is not. Researchers typically choose one dimension of an important array as a reference to the problem's size. The size is important because it effects the communication costs and the number of loop nest iterations for the parallel solution. Rather than trying to establish a technique to estimate the size of the problem, we simply consider the problem size in terms of its main components: the timing requirements for computation and communication.

We measure properties of the source code of an application, such as the number of loop iterations, the number of communication periods, and the volume of messages sent. We use these measurements to estimate the ratio of computation time to communication time for an application. We propose that this ratio can be used to form a metric which, when combined with information gathered about the target NoW will permit us to make important decisions

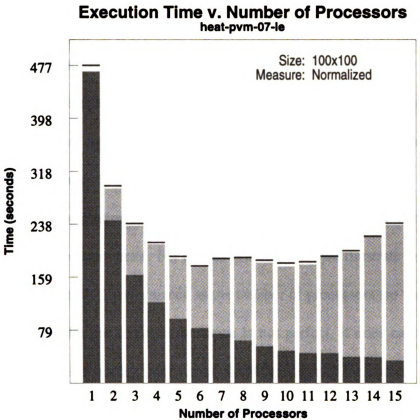

Figure 5.1: typical performance curve for parallel processing

within a restructuring compiler framework. We address one specific decision in this chapter — determining the fewest number of physical processors that minimizes execution time.

Consider an application that runs on a specific NoW. If the user fixes certain variables in the application that dictate the problem size, then based on the number of processors the compiler chooses, the compiler knows at compile-time how many loop iterations will be executed, how many messages will be sent, and the volume of messages transmitted per

process. Assuming the model has been calibrated for the specific NoW, we now have a vector of coefficients that parameterize the performance of the different components of the NoW.

By supplying the compile-time measures and the vector, our metric will produce a value proportional to the execution time as the number of processors varies. Using results from a specific application described later in the chapter, we can compare the performance metric to the measured execution times of the actual application. The measured times are obtained by running this application 15–20 times for each of the specified number of processors and averaging the execution times. For a fixed problem size, we illustrate the actual execution times and the predicted performance as the number of processors are varied in Figure 5.2. Note that while the predicted performance is not perfect, it does capture the first-order effects and reveals important information about the shape of the curve.

In this chapter, we show that the predicted performance curve can be determined prior to execution and used to improve the compiler transformations. One specific technique is to use the predicted curve to have the compiler choose the fewest number of workstations needed to minimize the execution time. Several assumptions, which are explored in 5.2, are required to take full advantage of the techniques described. These assumptions are commonly employed in current performance prediction research papers [26, 25, 13, 22]. Note also that the techniques described here have been performed manually on the source code for testing purposes — a compiler has not been implemented to automatically add the instrumentation.

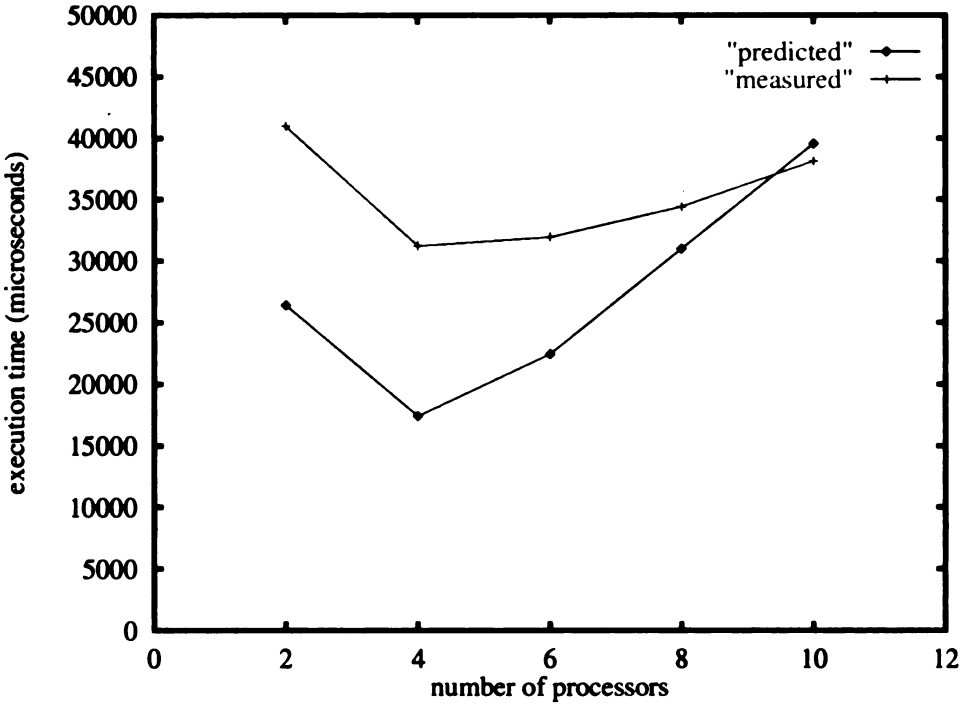

Figure 5.2: actual execution times and the predicted performance

5.1 Experimental Set-Up

To calibrate and test our model-based metric, we developed four data-parallel applications. We chose to use complete applications rather than a set of artificial or synthetic loops to calibrate our function. By doing so, we are able to capture the effects of intra-application network contention, the influence of medial I/O, and a natural mixture of loop nests. Sets of synthetic loops usually attempt to address the last issue but rarely do they address the first two. By taking this approach, we allow any application to calibrate the system, which may be useful in specific production environments. Below is a brief summary of our data-parallel system, the test applications, and the hardware platforms. A more detailed description of our system and the test applications appear in Appendix B.

5.1.1 Data-Parallel System

To implement the test applications on a NoW, we developed a simple C++ data-parallel library to interact with PVM 3.3.11. The library has classes to handle distributed arrays, spawning and managing of SPMD processes, group/neighbor communications, and timing the gathering of timing information. Like most data-parallel systems, our system uses static scheduling based on the owner-computes rule.

The library facilitates the instrumentation and calibration of the programs with a timing class. All of the applications were compiled manually because, in order to make the necessary changes automatic, we would require access to the source code of the data-parallel compiler, which we did not have while conducting the experiments. The library is unobtrusive. It adds one collective communication which appears at the end. Time spent in the support library is neither counted towards computation nor communication times. From the overhead shown in the graphs, it is clear that the library is not a significant factor in the overall execution time.

5.1.2 Applications

We used four complete applications to test our work and calibrate the metric. All of the programs were originally written as ordinary serial applications which were then converted to parallel, SPMD programs.

- Heat Transfer Application (heat)

  The first application calculates a sequence of images (a video) that indicates the transient heat transfer through materials with different thermal coefficients. Thus, an

engineer can observe the heat transfer through the design. Unlike many video applications, this application does not have frame-level parallelism because frame $(i-1)$ is needed to calculate the i^{th} frame. So, instead of performing each frame calculation in parallel, we distribute the columns of the array representing the image (and other associated arrays) across the processors. Since these slices are not independent, a data-parallel compiler, such as FORTRAN D or HPF, will generate the necessary communication, which we manually insert. A higher level compiler with automatic data decomposition would likely produce similar communication patterns. This application includes a number of different loop nests involving multiple distributed arrays, I/O procedures that gather individual frames to be written to disk, and messages to move columns of data between neighbors and global reductions.

- Fingerprint Matching Application (fprint)

  The second application searches a database to match the features of an unidentified fingerprint. In previous stages of the identification process, fingerprint features are extracted from digitized images and stored in a database. Since two digitized images of the same fingerprint are unlikely to have precisely the same features, a "scoring" function determines how closely the features match, which requires a global, distributed sort.

- Texture Segmentation Application (gbank)

  At another stage in the automatic fingerprint identification process, features are extracted from a digitized image. One technique uses a bank of Gabor filters to extract

the feature, which involves a series of convolutions. Besides feature extraction, Gabor filters are also used in other applications such as texture segmentation.

- Spatial Decomposition Technique (sdt)

  The fourth application applies a spatial decomposition technique (SDT) to the electromagnetic analysis of electrically large objects, such as various geometries for antennæ. The straight-forward formulation of the problem leads to densely-filled, numerically-intensive, complex-valued matrices. The SDT allows total operation count to be reduced by introducing subobjects which are simpler to compute. Calculations local to the subobjects are readily performed in parallel, but the subobjects need to synchronize frequently as the whole system converges to a steady-state solution.

5.1.3 Hardware Platforms

Part of the challenge of compiling for NoWs is the varied nature of NoWs. The networks can be configured with different speeds and protocols, and may or may not use a bus-based medium to communicate. To capture this variability, we ran our experiments on three platforms. All of the platforms were homogeneous and every workstation had a single processor running Solaris 2.5.1. Each platform is described below.

- ATM

  The first platform is a NoW with 12 Sun Ultra SPARCs connected by a Fore ATM switch. Our data-parallel system does not use the ATM API because an unreliable

service is inappropriate for parallel computing. In its place, we use the vendor-supplied TCP/IP interface, which is not as fast as the ATM API, but is more suitable for our tests. The designation `fa` is used in the figures to represent this system because it is the name of the Unix device driver. The bit rate of the network is 155 Mb/s per workstation.

- Fast Ethernet

  Another NoW configuration we tested has 12 Sun Ultra SPARCs connected by a Fast Ethernet switch. The bit rate is 100Mb/s per workstation. We use the designation `le2` for this system (from the Ethernet device driver `le` and the magnitude of the bit rate 10^2).

- Ethernet

  The last system has 12 Sun (40 Mhz) SPARC 10's connected by Ethernet. These machines are connected by a switch that gives a 10 Mb/s bit rate to each workstation. Our designation for this network subsystem is `le1`.

5.2 Instrumentation and Analysis

In addition to the serial-to-parallel transformations we perform on these applications, we also instrument our code to gather timing and count information. The former is a run-time measure used in calibration. The latter is a compile-time measure of loop iteration and messages. We instrument all of our test cases, but in a production compiler, only applications used for calibration would be fully instrumented. For applications not used for

calibration, we follow the same procedure described below, without modifying the code, to reveal the loop counts.

While it is theoretically undecidable to ascertain the number of iterations prior to execution, we offer approaches and simplifying assumptions to make the problem tractable for a broad class of scientific and engineering problems. Crucial to any reasonable model of the communication/computation ratio is an understanding of the phenomenon we call *observation skew*. We describe this phenomenon in 5.2.4 and show how our model adapts to it.

5.2.1 Compile-Time Measurements

The measurements we make prior to executing the code include a count of the number loop iterations, the number of communication periods, and the number of bytes transferred. Since the compiler inserts the communication primitives, it is simple to count any single message the compiler inserts. The number of times that message is transmitted, though, depends on the number of loops that enclose it. The number of loop iterations is more difficult. Determining the number is what we concentrate on here. We start by describing the counts for loops and then how to extend the counts over procedure boundaries. We describe some limitations that can be addressed by known compiler transformations and some that cannot. This discussion leads into the statement of our assumptions.

For a single loop, the count is calculated directly from the bounds and step size. To combine two loops, the iterations are summed if the loops are executed sequentially. They

are multiplied if the loops are nested. Loops within procedures are similar to nested loops but require global analysis.

To perform the global analysis, we build an interprocedural loop nest tree to count iterations for the whole application. We start at the leaves of the tree and work our way toward the root. A loop nest tree is the ideal structure. However, some calling sequences will form structures that are not trees. If a procedure is called in more than one place, then the interprocedural structure becomes a directed acyclic graph. Under these circumstances, a call site is chosen arbitrarily and the call parameters from that site are used. If there is recursion, the interprocedural loop tree becomes a structure with cycles. While it is still possible to count iterations with some recursive calls, that is beyond the scope of this work.

Difficulty arises in the model when loop bounds are not immediately available. Constant folding can be used to propagate values, when needed, to determine individual loop bounds. The iterations of `while` loops with simple expressions and simple induction variables can be counted by induction variable recognition methods. But even with these well-known compiler techniques, the determination of an iteration count is not always possible. Because the dimension of an important array may be an input to the program, it is unknown at compile-time. Based on the input data, branches in the loop nest tree may be executed conditionally, producing a varied effect on the iteration counts. Also, many numerical methods require looping until a tolerance is met, which means the number of iterations is not known until *after* the loop has been executed.

5.2.2 Assumptions

The limitations just mentioned above restrict use of our metric but they are not as severe as they may appear. The compiler can easily maintain a symbolic expression of the number of iterations. If these symbols are constant or are set once during execution, the metric can still be useful. Symbolic comparisons can be made between loop nests within the program. If the variables are constant and the transformation framework resolves the transformations to a binary decision, such as "12 processors or 24?", both versions could be placed in the object code and the appropriate one selected at run-time. The expression at the top of the loop nest tree describes the number of iterations in the whole program. Critical variables unknown at compile-time can be presented to the user with the statement, "specify these variables and the compiler will perform advanced transformations" (a/k/a partial evaluation). When ascending from a conditionally executed branch of the loop nest tree, one estimate is to assume the branch is always taken. For mutually exclusive branches, the maximum of all branches is a reasonable estimate. These worst-case decisions will be consistent in the calibration and use of the model.

The efficacy of the simplifications described above were not tested because they did not appear in our test cases. From this point, we will tacitly assume that we can count the loop iterations.

We also assume that we have exclusive access to the NoW — workstations and local network. The workstations might be used periodically as general time-sharing machines but we assume that when we are calibrating or running as a NoW, that all other users are denied access. Although we have not documented it formally, our experience has led us

to conclude that our techniques would fail if users shared the machines during our experiments.

5.2.3 Run-Time Measurements

To fit our model to a specific NoW, we need to gather performance information about the NoW. We gather this information by instrumenting an application and executing it on the platform. One run of the application generates a record with several pieces of data. We keep track of the date (in the usual internal Unix style: the number of seconds since January 1, 1970), the number of processors involved, the size of the application, the average size of a message, and for each processor, the time for computation, time for communication, and the overall time for the entire application. We also keep a precise count of the iterations per processor, corresponding to each of the three time measurements. There is also a provision for distinguishing individual loop nests within the application but that feature is not used in this set of experiments. The pertinent information is stored locally on the workstations in a small data structure until the application ends. At the end of the run, all of the data is gathered on one machine and a single record representing the run is output. One file is used to store all of the records related to a single configuration. The filename encodes the application and version number, the communication protocol, and the network interface.

As an example, consider one record, shown in Figure 5.3, generated by the heat application running on the ATM network of Ultra SPARCs. This record is in a file named `heat-pvm-08-fa.data` which indicates the network, the current version of the heat program (08), and the base communication system in use. We have utilities to parse and

```
#date: Mon Jun 23 04:34:39 1997
867054879 6  200 encl     { # 27.385775 avg msg size
        77267/28158 185804/46159244 270215/8 (00)cerium
        102903/28167 154323/46084844 263899/8 (03)lithium
        102434/28167 155294/46084844 263924/8 (02)helium
        95123/28167 155367/46084844 257397/8 (04)calcium
        95514/28158 162016/47479112 263924/8 (01)magnesium
        82671/28167 173494/45204932 263769/8 (05)californium
}
```

Figure 5.3: one timing record

process the files with records in the form of Figure 5.3. The '#' indicates a comment that runs to the end of the line. The last column in the fields (within braces) is the workstation name and its logical process number. Process 00 is the I/O process and master. The process to the right of the master is 01.

5.2.4 Skewed Observations

Measuring communication and computation times for the *system* requires careful attention because the measurable run-time aspects of the system can give misleading results. Our instrumentation measures the computation by noting the time at the beginning of a loop nest and at the end. The difference is accumulated for the entire execution time and the iterations are recorded. The communication time is measured in a similar way — the time is noted at the beginning and the end of a section of communication code. The number of messages and the size of the message is also recorded. Consequently, a program appears as a sequence of alternating communication and computation periods. Martin *et al.* [48] studied the effects of latency, overhead, and bandwidth in a NoW. One of their results is

that applications are most sensitive to overhead and even if latency is completely hidden by overlapping of the transmission time of messages with computation, the overhead will still cause an application to appear to alternate between periods of communication and computation. Thus, whether the compiler can or cannot overlap messages is unimportant to our model. If the compiler does overlap messages, we are simply modeling the overhead.

These measures accurately represent the delays that the individual processes observe, but these numbers do not accurately measure the time spent actually doing the computation or communication. For computation, the operating system may interrupt the process so that computation *delay* is longer than the actual computation time. For communication, the time doing the communication may not even be the most significant portion of the communication delay. Specifically, the communication delay will include at least three factors: waiting for the corresponding process, the actual communication time, and the time that the process may sit in the ready queue waiting to be rescheduled. In Figure 5.4, we illustrate these factors for two processes. Note that in the two-processor case, one process will finish before the other and the t_1 factor will only be reflected in one processor's delay. Also note that the nondeterministic t_3 factor will occur for every message since sending a message always deschedules a process in general-purpose operating systems. Consequently, each processor observes different delays during a single execution of an application. Furthermore, this analysis assumes perfect load balancing. Even a slight load imbalance will make skew more pronounced.

Incorrectly summarizing these multiple views of the system leads to statistically unexplainable fluctuations. We call this phenomenon *observation skew*. We take a pragmatic

approach to masking its effects. Our concern here is how to produce a single number to represent the whole system communication time or the whole system computation time.

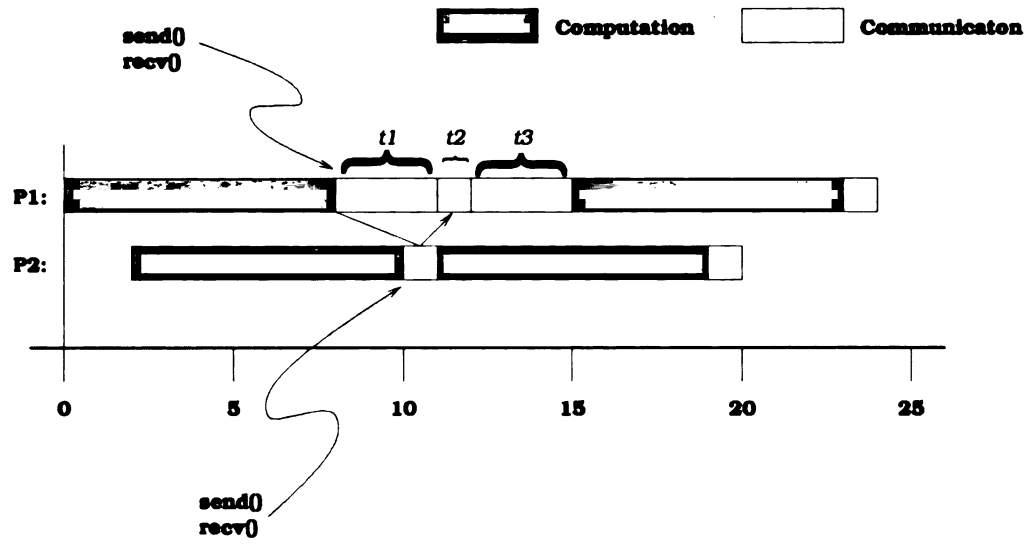

Figure 5.4: skewed observations of communication delays

We considered three ways of addressing the skew issue. Let m be the number of processors in sample i and, for $1 \leq j \leq m$, $t^{(i)}_{\text{comm},j}$ be the j^{th} processor's observation. One solution is to choose the maximum,

$$t^{(i)}_{\text{comm}} = \max_j \left\{ t^{(i)}_{\text{comm},j} \right\} \quad \text{(communication)}$$

$$t^{(i)}_{\text{comp}} = \max_j \left\{ t^{(i)}_{\text{comp},j} \right\} \quad \text{(computation)}$$

$$t^{(i)}_{\text{comb}} = \max_j \left\{ t^{(i)}_{\text{comb},j} \right\} \quad \text{(overall)}$$

but this leads to inconsistencies (for example, $t_{\text{comm}} + t_{\text{comp}} > t_{\text{comb}}$ is possible) that are hard to model. Another approach is to look at one processor. In our application, the first task created is responsible for starting all other tasks and the handling of the I/O. It

will always be the final task running. The processor running this task is a likely candidate because its overall time is what the user observes, which in a sense, makes it the "truest" overall time. But its computation time might not represent the system well because of load imbalance. Also, this process has more responsibilities in collective communications. A third technique uses a *normalized* observation that amounts to a weighted average,

$$t^{(i)}_{\text{comm}} = \max_j \left\{ w^{(i)}_{\text{comm},j} \right\} \cdot \sum \left(\frac{t^{(i)}_{\text{comm},j}}{w^{(i)}_{\text{comm},j}} \right) \quad \text{(communication)}$$

$$t^{(i)}_{\text{comp}} = \max_j \left\{ w^{(i)}_{\text{comp},j} \right\} \cdot \sum \left(\frac{t^{(i)}_{\text{comp},j}}{w^{(i)}_{\text{comp},j}} \right) \quad \text{(computation)}$$

$$t^{(i)}_{\text{comb}} = \max_j \left\{ w^{(i)}_{\text{comb},j} \right\} \cdot \sum \left(\frac{t^{(i)}_{\text{comb},j}}{w^{(i)}_{\text{comb},j}} \right) \quad \text{(overall)}$$

where $w^{(i)}_{\text{comm},j}$ represents the number of iterations for processor j, $1 \leq j \leq m$. This takes into account both sources of skew. We are satisfied with its ability to model the state of the system because from experience we have found that a simple average is best for the overall times. For the computation, we used a normalized average. For communication, we have found the I/O node's time to be the best number to summarize the system's state. Otherwise, imbalances in the workload created fluctuating statistical properties that made it difficult to model the system.

5.3 Formulating a Metric

By developing a metric, we are analyzing the relationship among several variables, including execution time and compile-time measures. This is called regression analysis, a well-known statistical tool. To use the regression equation, we need to check that our data meet the standard assumptions for regression analysis. Namely, we want to show that the two main components of our metric, computation times and communication times, for a NoW approximate a normal distribution. Also, to effectively use the model, we want to show the practical result that the variance of the overall execution times are small so that it is not likely a random perturbation will render our results meaningless. Following those checks, we develop a general performance metric based on a model where the explanatory variables are compile-time measurements. Performance will, of course, depend on the specific platform. We incorporate platform-specific information when we calibrate the model in 5.4.

5.3.1 Statistical Tests

Below, we show that distribution of execution times of a program of fixed size and number of processors is approximately normal. We calculate an interval for the variance such that we are 90% confident the true variance falls within the interval. Strictly, we know that execution times are not normal. There is an absolute minimum time that gives a left-hand boundary to the probability distribution function. This asymmetry violates a Gaussian distribution. Nonetheless, by performing these tests, we are showing that the actual distribution is statistically close.

We start with the hypothesis that variations in computation and communication times follow a normal or Gaussian distribution. We test this hypothesis by fixing the problem size at 300 and looking at the actual distributions of a large number of executions (21 to 37 runs each) when the number of processors is 2, 6, and 12. Measured execution times are grouped into five regions and the frequencies for each region are graphed in Figure 5.5 and Figure 5.6 for the ATM platform and the Ethernet platform, respectively. The solid blue line shows a normal distribution with the mean and variance determined by the sample mean and variance. The red lines form a histogram of the relative frequencies for the five regions.

We test our hypothesis that computation and communication times are normally distributed using a χ^2 goodness-of-fit test. The samples are divided into five regions. Since every region needs at least five samples for the test, adjacent regions were merged so that the number of samples in each region are greater than four (leaving 3 degrees of freedom in each case). Then, we compare the observed frequencies of execution times per region with the frequencies expected for a normal distribution. This approximates a chi-square distribution with $\nu = 3$ degrees of freedom. By using the critical value $\chi^2_{0.05} = 5.991$, we can reject the null hypothesis with 95% degree of confidence and conclude that the fit is good. The calculations for each population (the platforms and the number of processors are varied — each configuration is a population) are summarized in Table 5.1. Every population tested passed the test for computation and for communication. From the graphs, though, the distribution of communication times appears slightly asymmetrical, as we would expect.

Once we accept that we have a normal distribution of computation times, it is desirable to know the accuracy of the sample variance. To determine this, we used the χ^2 distribution

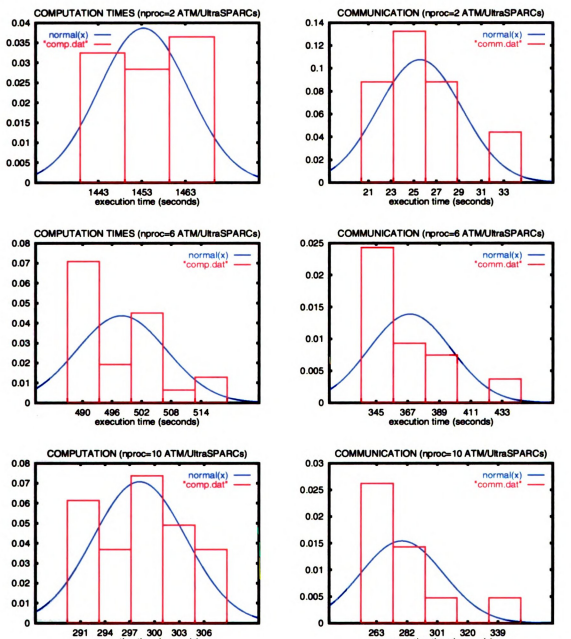

Figure 5.5: histograms of measured communication/computation times and Gaussian pdfs

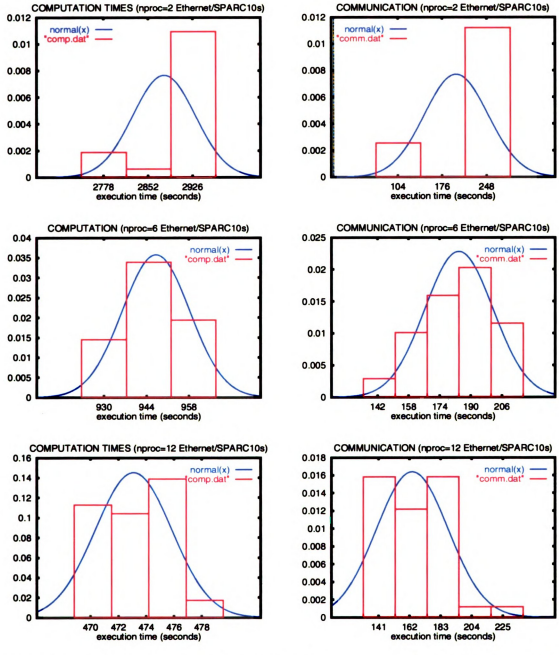

Figure 5.6: probability distributions for the same size problem with different numbers of processors on an Ethernet

Table 5.1: summary of χ^2 goodness-of-fit tests

χ^2 for ATM/UltraSPARC Platform
COMPUTATION

Processors	n	Mean	Std Dev.	ν	χ_2	Result
2	24	1453s	10.3s	3	4.010	pass
6	24	498s	9.1s	2	0.246	pass
10	21	298s	5.6s	2	0.371	pass

COMMUNICATION

Processors	n	Mean	Std Dev.	ν	χ_2	Result
2	24	25.6s	3.7s	2	0.8853	pass
6	24	369s	28.7s	2	1.863	pass
10	21	279s	25.8s	2	2.536	pass

χ^2 for Ethernet/SPARC10 Platform
COMPUTATION

Processors	n	Mean	Std Dev.	ν	χ_2	Result
2	39	2876s	54.1s	2	0.283	pass
6	38	947s	11.8s	2	3.889	pass
12	37	474s	2.6s	3	5.016	pass

COMMUNICATION

Processors	n	Mean	Std Dev.	ν	χ_2	Result
2	39	197s	54.0s	2	2.867	pass
6	38	185s	18.3s	3	2.178	pass
12	37	295s	46.2s	2	2.772	pass

Table 5.2: summary of 90% confidence intervals for overall execution time

Platform, #proc.	measured	Standard Deviation	Avg. Execution Time
ATM/UltraSPARC, $n = 2$	7.6s	$5.71 < \sigma < 11.9$	3007s
$n = 6$	22.1s	$17.2 < \sigma < 31.3$	1116s
$n = 10$	29.8s	$23.4 < \sigma < 41.7$	790s
Ethernet/SPARC10, $n = 2$	146.2s	$113.3 < \sigma < 210.1$	1487s
$n = 6$	37.0s	$28.7 < \sigma < 53.2$	858s
$n = 12$	8.3s	$6.4 < \sigma < 12.2$	582s

again to calculate a 90% confidence interval for the variance (and subsequently the standard deviation). The results for the overall execution time are shown in Table 5.2. In each case, the range of the standard deviation is a small percentage of the average computation time. Thus, these results suggest that it is statistically unlikely that a fast two-processor sample will be faster than a slow four-processor sample. Without this assurance, any statistically based metric would be moot.

5.3.2 General Formula

As we have suggested, the general shape of the performance curve comes from the two major components, communication and computation. We treat both as functions of the problem size and the number of processors involved. Below we describe our metric and articulate our choices for the general scheme.

First, we consider the computation alone. For our SPMD data-parallel applications static scheduling is used. The iterations are divided among the processors based on the distribution of the data using the owner-computes rule. Thus, if the number of distributed elements is evenly divided by the number of processors, and every element is updated inside the loop, then the work is evenly divided. If the number of elements is not evenly divided by the number of processors and every element is updated inside the loop, then one or more processors will do an extra iteration to make up the deficit. When none of the elements are updated inside the loop, the load balance is determined by the application. The last case is important because it can be a major cause of skewing. Using the worst-case, this leads to the simple approximation of w/p iterations, where p is the number of processors and w is the total number of iterations. Since the execution time is nonlinear in terms of p in this equation and the mathematics is simpler with linear functions, we immediately transform this expression to the iterations per process, or x_1, which can be directly measured at compile-time. For programs we are modeling, we will assume that every iteration takes approximately β_1 time units, so our cost function of computation is

$$t_{\text{comp}} = \beta_1 x_1$$

The assumption that all iterations are the same is clearly false. But we are willing to make this assumption because all programmers tend to write loop nest bodies that are of the same order of complexity. It is unlikely that one loop body will be 100 times or 1000 times bigger than another loop body in the same program. Furthermore, we are more concerned with the shape of the function, not the specific execution time. That is, we will be making

relative, not absolute, judgments with this function. As demands increase for more precise functions, this may need to be investigated as noted in the future work in Chapter 6.

Estimating communication is not as simple as computation. The effects of a message are less deterministic because they require interaction with the operating system (and possibly the scheduling algorithm of the operating system) and interaction with the network interface. The type of communication is also a factor. A common type of communication in a data-parallel solution moves columns of data between neighbors. Ideally, this communication cost should be independent of the number of processors, but in practice it is not. If the processors are connected by a bus-based network, as is typical with traditional Ethernet, then contention for the medium would certainly be an issue and a source of nondeterminism. Another type of communication is global reductions. The cost of a global reduction may increase logarithmically or linearly (with respect to the number of processors), depending on the implementation. Our system is not logarithmic. We considered a number of models which were deemed inadequate. Several failures are documented in the following section. Originally, we approximated communication costs by counting the loop-induced communication periods, per process. Our reasoning was that communication costs are dominated by message startup times. We speculated that that would be sufficient. Instead, we found that the number of messages *and* the volume of messages, per process, was a better estimate. If we included the number of processors involved, to account for bus-based networks and global reductions, then we had a slightly better approximation. So our approximation of communication time is

$$t_{\text{comm}} = \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4$$

where x_2 is the number of messages, x_3 is the volume of messages, and x_4 is the number of processors involved. The constants, β_i, are the relative proportions of each factor. We discuss these more fully in 5.4.

Since the compiler inserts the messages, it knows which loops enclose it and the message sizes. Thus, the number of messages and the average size are kept in two scalars while traversing the loop nest tree at compile-time.

Our cost function is the sum of the communication and computation costs in the loop nests of the program. Written as a single equation, y is

$$y = \underbrace{\beta_0}_{\text{overhead}} + \underbrace{\beta_1 x_1}_{\text{computation}} + \underbrace{\beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4}_{\text{communication}} + \underbrace{u}_{\text{nondeterm.}} \tag{5.1}$$

where x_1 is the number of iterations of computation, x_2 is the number of messages, x_3 is the volume of the messages, and x_4 is the processors. The constants β_i represent a relative weighting of the factors. All activity outside of loops is represented by the lone constant β_0 and the u term is a random variable introduced to represent the nondeterministic delays — factors not modeled. Finally, y is a dependent variable proportional to the execution time. All of the explanatory variables (x_i's) are functions of the problem size and the number of processors. Thus, we have a function that meets our requirements for a transformation framework metric. In the next section, we describe how to determine the β_i's and how we tested the metric.

Although these two components were established separately, strictly speaking they are not always necessarily independent. Some compilers will transform loops such that messages overlap with computation and this relates variables x_1 and x_3, a violation of an as-

sumption made in the next section. We believe this to be a minor effect that does not disturb the first-order effects we are attempting to model. The results that follow support this belief.

5.4 Calibration and Experimental Results

The function previously described is not complete; the β_i's must be specified. In order to use an empirically-based metric for a NoW, it is necessary to make measurements on the active system because, clearly, a single program will perform differently on different platforms. We call the process of determining the β's for a specific platform *calibrating* the metric.

The calibration stage involves running an instrumented application multiple times with different parameters for the number of processors and the size of the problem. At the end of each execution, the timing results for each process are gathered and written as a single record to a file (see 5.2.3). For development purposes, we execute several runs so that for each problem size and number of processors combination, we have 12 samples. In this section, we use these samples to determine the specific β constants of Eq. 5.1. These constants relate the performance of the major subsystems in a specific NoW system. To test this, we designed a number of experiments to check the sampled data. We combined the general formula and our sampled data to determine the constants using the standard approach of multiple linear regression. Next we examined suitability of our metric and the adequacy of the model. We focus on two standard goodness-of-fit tests. First, we look at the proportion of total variability. Second, we inspect the graphs of residuals against the

important variables in the model and system. This helps to establish the homoscedasticity condition necessary to use linear regression. We also point out the inadequacy of some less specific models.

5.4.1 Multiple Linear Regression

To generate a collection of samples, we execute the instrumented application twenty-four times varying the size of the problem (100, 200, 300, and 400) and the number of processors ($n = 2, 4, 6, 8, 10, 12$). We call each combination a *configuration*. A collection of all possible configurations is called a *run*. This produces samples of various ratios of communication and computation. We perform several runs to generate twelve samples for each configuration.

For each sample, we calculate a normalized overall time and store it as an entry in the vector y. From the compile-time measures, we can also obtain the computation (x_1), number of messages (x_2), the volume of the messages (x_3), and the number of processors (x_4). If we have k executions of the application, we can arrange these data in a $k \times 5$ matrix, X, where each row of data comes from one execution. (Note, the first column of X is all 1's and corresponds to the constant β_0 in Eq. 5.1. In terms of our model, β_0 is the total overhead outside the loop nests that occurs once in every sample.) Our problem of determining b is a matter of solving

$$Xb = y \tag{5.2}$$

Since one run generates twenty-four samples, it is unlikely that all the samples will perfectly fit the model. The rank of the matrix will certainly exceed five. Thus, we have an overdetermined system and the best we can do is to choose a vector $\bar{b}$ to approximate the solution. Common practice in this case is to minimize the squares of the errors (least-squares solution), where the errors (called residuals) are defined as

$$e = y - X\bar{b}$$

Geometrically, this means projecting the samples y onto a 5-dimensional subspace defined by $\bar{b}$. We want the projection that minimizes the distance. This will be a line orthogonal to every column of X,

$$X^{\top} e = 0 \qquad \text{or} \qquad X^{\top}(y - Xb) = 0$$

or

$$X^{\top} X\bar{b} = X^{\top} y \tag{5.3}$$

Our calibration program reads from the data file of samples and stores them in X and y. We form a 5×5 matrix $A = X^{\top} X$ and a new vector $c = X^{\top} y$. Since A is formed from $X^{\top} X$, it is always symmetric, which means there is a stable, fast decomposition of A:

$$A = LDL^{\top}$$

where L is a lower triangular matrix. Thus, we solve for $\bar{b}$ by substituting A and c into Eq. 5.3 to get

$$LDL^{\top}\bar{b} = c$$

By doing a forward substitution, dividing by the diagonals, and a backward substitution, we can quickly solve for $\bar{b}$ which is also the solution to the vector $\bar{b}$ in Eq. 5.1. This is our calibration vector. The independent variables (x) are measured at compile-time. The calibration is done once for a configuration ($\bar{b}$ is calculated), and y is determined by these two:

$$y = \bar{b}^{\top} x$$

In terms of compiling for a NoW, the vector $\bar{b}$ represents the performance relationship between the communication and computation components of the workstations. The vector x represents factors measured in a particular application. The resulting value y can be used in comparisons with other y's for the same application but with different problem sizes or numbers of processors. Below, we test the accuracy of the model.

5.4.2 Adequacy of Model

Two steps remain. First, we need to establish that the model adequately describes the sample points. This alone gives an important result: an application can be used to calibrate the model and then the model will predict that application's performance (for other configura-

tions). The second step extends the result to show that the calibration with one application can be used to predict the number of processors required to minimize the execution time of another program.

We test the adequacy of the model by examining the two major components separately, lest we risk the case that one component's good fit masks the other's poor fit. If each component fits its part of the model and the whole model fits, then we can be relatively sure that we have chosen an accurate model on which to base our metric. We also show that models which do not include as many factors, do significantly worse.

Although we discuss only one application (heat) on one platform (ATM/UltraSPARC) throughout this subsection, we summarize the results of the same tests for all the applications and all the platforms at the end.

To test the computation component, we simply use the number of iterations factor, x_1, and a simple linear regression equation for computation:

$$t_{\text{comp}}(x_1) = \beta_0 + \beta_1 x_1$$

First, we calculate the coefficients (the $\bar{b}$ vector) for this equation such that the sum of the squares of the residuals is minimized ($\bar{b} = (X^\top X)^{-1} X^\top y$), where y is the measured computation times and X is the iteration counts. With these constants, we graph t_{comp} and the actual samples in Figure 5.7. As can be observed, the model is close at both ends of the scale. Despite a few outliers, the data support the obvious relationship: computation time is proportional to the number of iterations. Note that in this model, β_1 represents the average time it takes to execute one loop iteration. This time will vary from application to

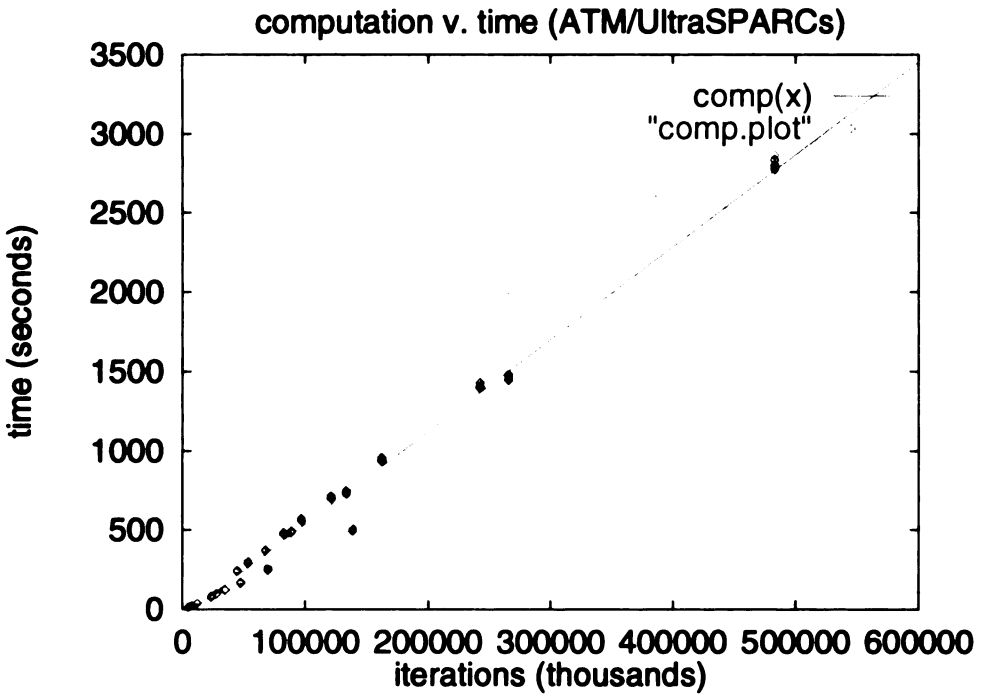

Figure 5.7: number of iterations v. computation time

application, but our use of the model tolerates a wide variance in this coefficient. We also consider the other coefficient, β_0. For our working example, it is very small ($\beta_0 = -46.120$ μs). When we take the model to the extreme case $x_1 = 0$, the computation time for zero iterations is appropriately naught.

Visual inspection will detect gross model violations, so it is a useful and necessary step. However, it alone is not sufficient. A more formal means of judging the adequacy of the model is to assess the square of the multiple correlation coefficient, R^2,

$$R^2 = 1 - \frac{\sum (e_i)^2}{\sum \left(t_{\text{comb}}^{(i)} - \mu_{\text{comb}} \right)^2}$$

which is the proportion of total variability for simple linear regression. We will also use it as a goodness-of-fit test for multiple linear regression. When errors are small compared to a simple estimate of the mean, R^2 approaches 1.0 and indicates a good fit. An R^2 close to 0.0 suggests no relationship between the model and observed samples. Along with the visual inspection and an analysis of residuals,[2] the proportion of total variability makes a fairly strong case for the adequacy of the model. The proportion of total variability for the computation component of our working example is $R^2 = 0.9858$ (based on 361 samples). This is interpreted to mean we have described 98% of the variability in the data.

For the communication component, the model is not as simple. A number of models were tested and found to be inadequate. Shown in Figure 5.8 is the distribution of communication times as measured by the communication delay observed by the I/O processor. We can see a fair amount of variation across the different configurations of processors and problem sizes. Note that when the problem size, which is related to the image resolution for the heat application, increases, the amount of computation time and communication time increase exponentially. This is not reflected in the figure because we do not calculate the same number of video frames in the heat application as the resolution increases. This is consistent with the way the application would be used and to do otherwise would result in an unmanageable range of execution times. A small problem size would be too short in duration to be statistically useful while a single execution of the large problem size would be too long in duration to make repeated tests practical.

While Figure 5.8 uses axes that are familiar, a more instructive graph shows the communication delay as a function of the number of communication periods and the volume of

[2] A thorough analysis of the residuals for the whole model is presented on page 102.

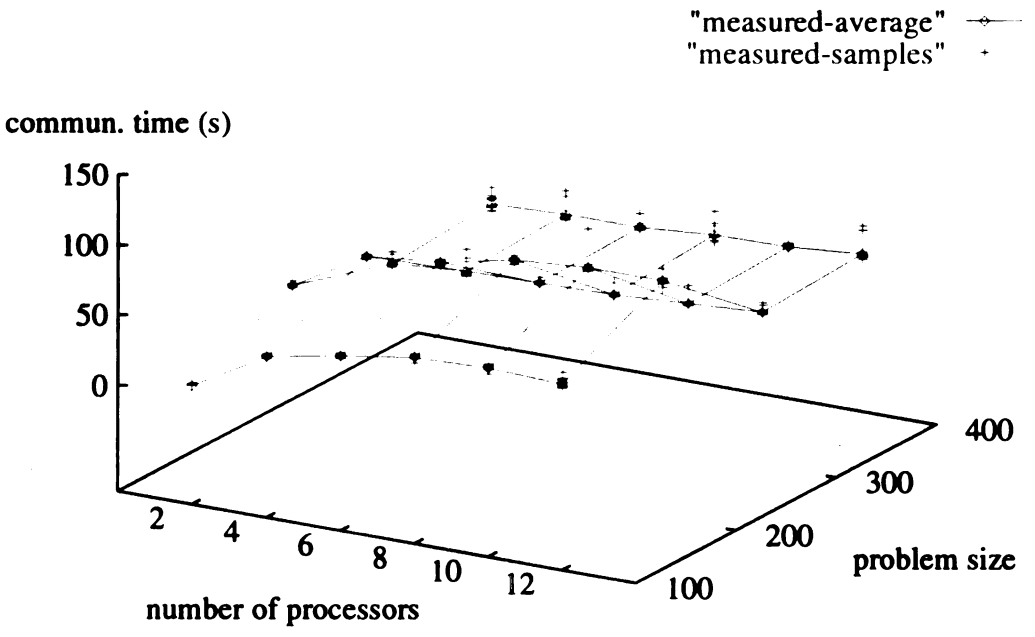

Figure 5.8: number of messages v. communication time

messages injected in the network. Figure 5.9 shows the individual samples and a measured average calculated per configuration. The measured average is a set of points where each point corresponds to an average of all of the samples of the same configuration. For example, one point is generated by averaging all of the communication delays for samples where the problem size is 100 and the number of processors is 2. Another point is generated for the configuration of size 100 and 4 processors. This is helpful for visually inspecting the fit. However, the measured averages are not used in determining the model coefficients.

The clear non-linearity around 750 thousand doubles is a concern. It highlights an important assumption in our model. It is well known that large messages are more efficient than small messages because of a start-up cost. But there are also other discontinuities in message time formulas for real systems. For some distributed processing systems, such as p4, large messages are handled differently than small messages. Often at the network interface, messages are split into packets, which may cause discontinuities as message sizes vary. One additional byte to a single message can require an additional packet. By assuming messages are related to just communication volume and the number of messages, we are ignoring many sources for discontinuities, such as message size. Despite the discontinuity, we still attempt to model the communication as linear, accepting the fact that there will be a loss of accuracy.

The result of fitting the data of Figure 5.9 is shown in Figure 5.10. The metric tends to be a little flatter but captures the essence of the curve. The multiple correlation constant, $R^2 = 0.7027$ suggests that roughly 70% of the variability is explained by our model. A more advanced model would consider factors such as individual message size which may address the unexplained 30%.

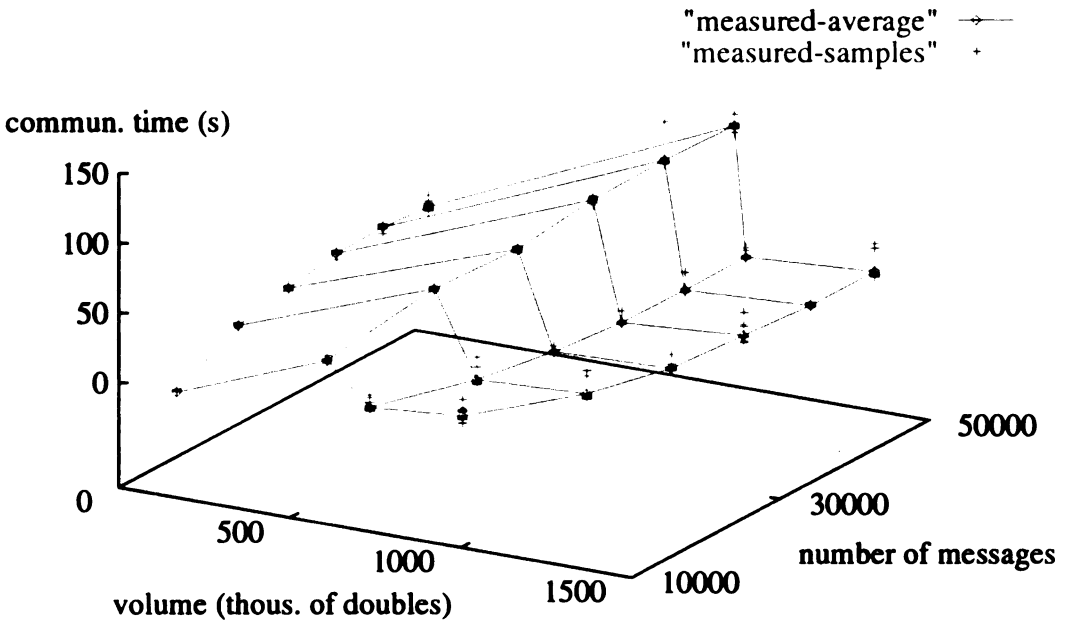

Figure 5.9: communication time v. message volume and number of messages

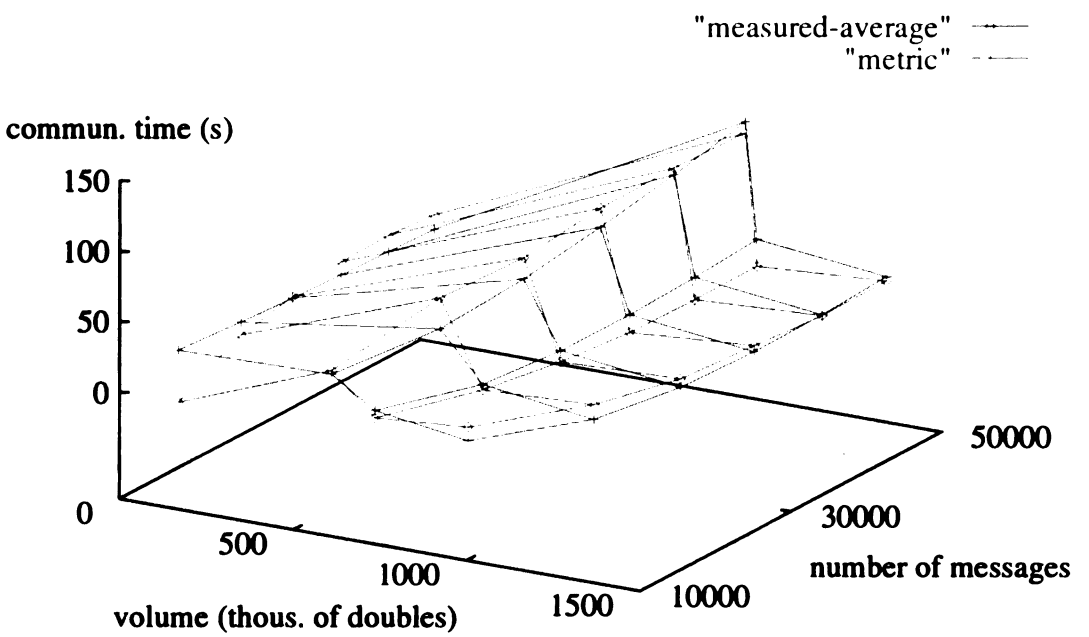

Figure 5.10: communication time v. message volume and number of messages

Now consider the whole model. Combining the two components and the general equation developed previously,

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4,$$

we can test the fit of the sample execution times. Note that we use the total execution time for the y values now, not the sum of the computation and communication times. Consequently, our β_0 is now the sum of the computation component's β_0, the communication component's β_0, *and* a constant factor that represents the general overhead of all the computation outside of loop nests.

Viewing the samples along all five axes simultaneously is not possible but we can graph the samples based on their configuration of number of processors and problem size. In Figure 5.11 we show the individual samples for the overall execution time and an average for each configuration, as we did with the communication component. After fitting the function to these samples, we show the measured execution times with the predicted execution times in Figure 5.12. Although the scale is very large and it is difficult to measure absolute errors from the graph, the graph shows that the general shape matches. A closer observation of a "slice" of this graph reveals that long execution times are fairly well matched, but the short execution times are not. With the multiple correlation coefficient at 0.9744 (based on 354 samples) for the overall model, we are reasonably confident our model is reasonably accurate.

As an important final check, we examine the residuals. Systematic variations indicate structure in the data that is not unaccounted for by the model. We graphed the residuals

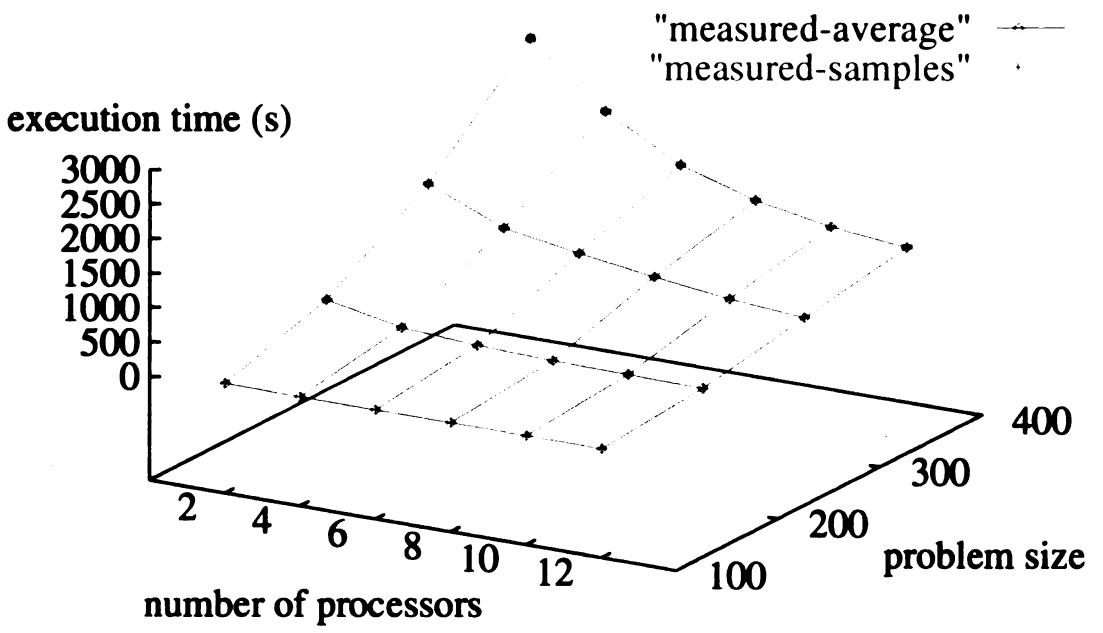

Figure 5.11: number of messages v. communication time

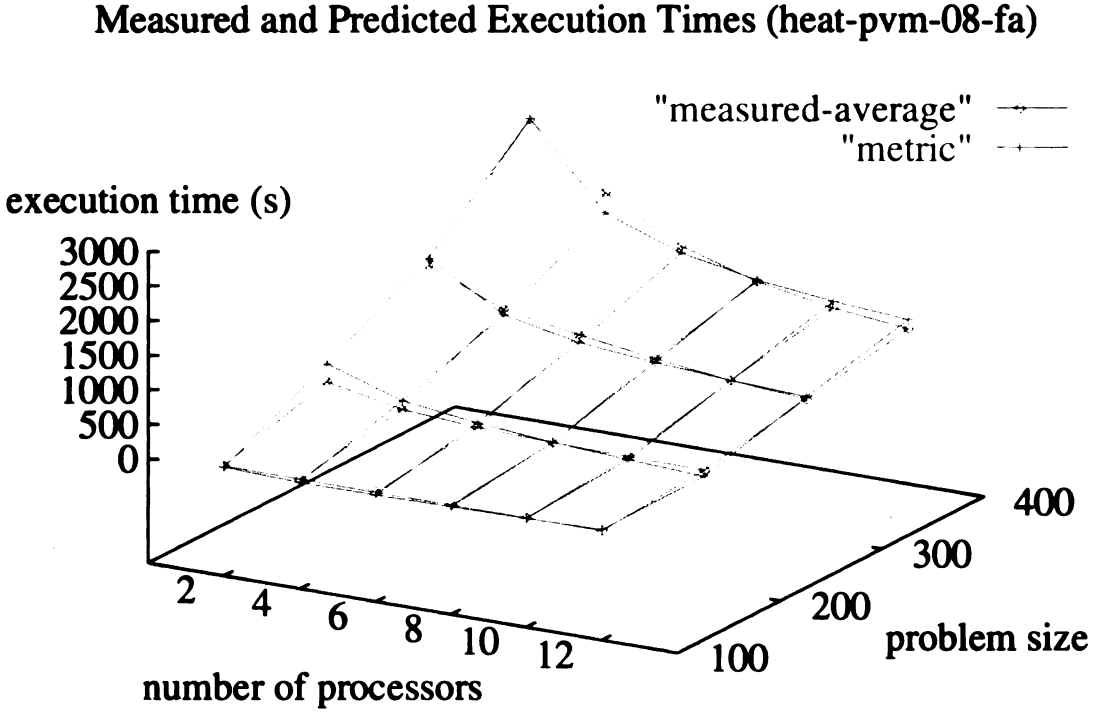

Figure 5.12: number of processors and problem size v. communication time

Table 5.3: several inadequate models of communication time

explanatory variables	R^2
num. of msgs (per processor)	4.05×10^{-5}
num. of msgs $\times$ num. of procs	3.39×10^{-5}
vol. of msgs (per processor)	0.0009919
vol. of msgs, num. of msgs	0.6314
vol. of msgs, num. of msgs, num. of procs	0.7172

against the four explanatory variables: the number of processors, the number of messages, the volume of the messages, and the number of iterations. We also considered several system variables. Since the number of iterations, number of messages, and volume of messages are related by the problem size, we also graphed the residuals against the problem size. Finally, we graphed the residuals against the dependent variable, execution time. The results are recorded in Figure 5.13. In our plots, the vertical alignment simply indicates that we have multiple samples for that x-coordinate.

To put the results in perspective, we present some "broken" models and the proportions of total variability in Table 5.3. These models omit some of the factors we ultimately decided to include and their R^2 value reflect the diminished accuracy. For example, using the number of communication periods alone to model the communication time did poorly (a very low R^2). This suggests that from the set of factors we chose to investigate, all are necessary.

All of the results thus far are for the heat application running an ATM network of Ultra SPARCs. We repeated all of the experiments for each platform and each application. The

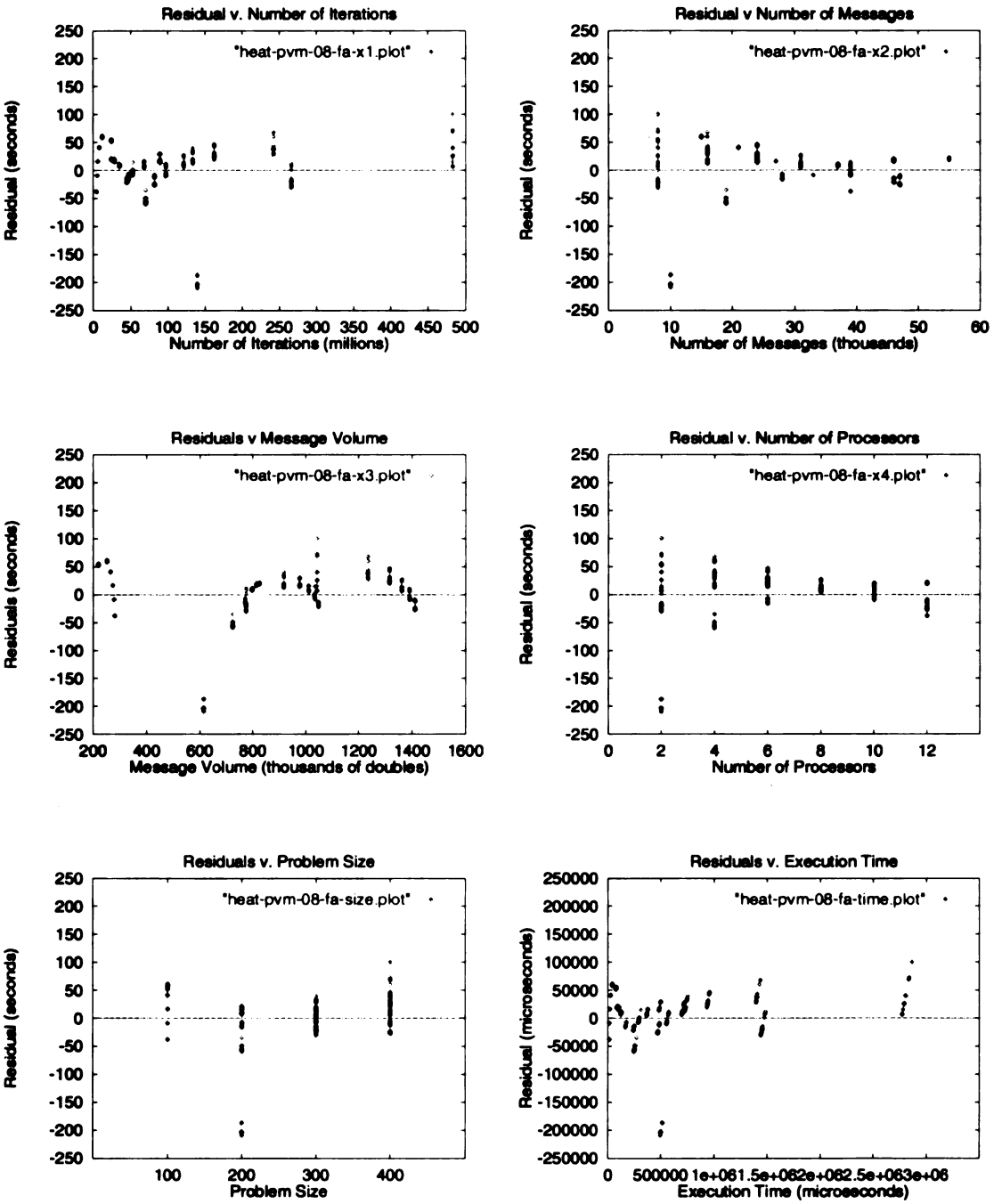

Figure 5.13: residuals plotted against explanatory variables

Table 5.4: results for each application on each platform

ATM network of Ultra SPARCs

Application	samp	R^2
heat	354	0.974399
fprint	338	0.997946
gbank	374	0.999972
sdt	580	0.983224

Fast Ethernet network of Ultra SPARCs

Application	samp	R^2
heat	342	0.975191
fprint	340	0.998786
gbank	370	0.999974
sdt	600	0.985542

Ethernet network of SPARC 10s

Application	samp	R^2
heat	321	0.998651
fprint	405	0.995880
gbank	260	0.999853
sdt	⋆	⋆

⋆ not available

results are summarized in Table 5.4. Note that the Ethernet of SPARC 10s NoW was out of commission when the last application was being tested.

5.5 Validation

The calibration and statistics of the previous section verify, in an abstract sense, that the model is behaving as we intended it to behave. The work described in this section is meant to validate that the model can be applied to our specific problem: guiding compiler transformations. This section is meant to assay the adequacy of the model. To produce substantial

results here would require the development of a transformation algorithm within some existing framework and then judgment of the metric and the algorithm's effectiveness. That is not our goal, however. We merely want to show that it is possible to use the metric and so we have chosen a modest test. We ask, "after calibration, can the metric predict the fewest number of processors needed to maximize performance?" To answer this, we have organized an experiment using our four test applications. For each platform, we use run-time information to calibrate a metric. Then we use this metric to predict the performance of the four applications. Specifically, we compare the predicted number of processors and measured number of processors that minimize execution time. These results are summarized in Table 5.5 for the ATM NoW, Table 5.6 for the Fast Ethernet NoW, and Table 5.7 for the Ethernet NoW. Note that the last NoW was not available when the fourth application was tested, so there are fewer results in this table. Also note that one of the machines in the other two platforms was unavailable which limited the number of processors to 10. A $\star$ is used to indicate when the minimum was mispredicted.

There are several features in the data presented that warrant discussion. First, note that every application did well at predicting itself. This may not be surprising but is in itself a useful result in a setting where the same application is used repeatedly. Second, many configurations identified 12 as the minimum number of processors because 12 is the maximum available in our system. Although the first three applications would not be considered "embarrassing parallel" there is still a performance advantage for them to use more processors if they were available. While the model correctly predicts this for most cases, it is, unfortunately, only a partial validation of our approach. The fourth application, SDT, was considerably more communication-intensive. For the smaller data sets, the performance

slowed down after exceeding a critical number of processors within our system's capabilities. Certain applications, (such as GBANK and FPRINT) when used to calibrate the model, did a poor job of predicting SDT's performance. On the other hand a model calibrated by SDT did correctly choose the maximum number of processors for GBANK and FPRINT. Certainly, we can conclude from this that some applications are unsuitable for calibrating the metric. But also from this, we speculate that SDT successfully predicts GBANK and FPRINT because it meets a minimum criterion that some of its samples have communication times that significantly influence the execution time. Until GBANK and FPRINT have enough samples where communication plays a significant part of the execution time, they will be unsuitable applications to calibrate the metric.

The last thing to consider when reflecting on the data presented is the consequence of a misprediction. It is the nature of multiple linear regression that there will be errors, but the errors are introduced such that they minimize the effects on the dependent variable. In our case, the dependent variable is execution time. Consider the case when HEAT's run-time information is used to calibrate a metric which is subsequently used to predict the other applications' performance. (Take, for example, the first row of blocks in Table 5.5.) The model mispredicts size 100 for HEAT. The model says to use four processors but we know from direct measurement that twelve minimizes execution time. In fact, a line tangent to the performance curve at four processors is approximately horizontal. Using the actual run-time measurements and the relative error in the number of processors, we have a fairly flat slope:

Table 5.5: results of minimizing the execution time for ATM

APPLICATION USED TO CALIBRATE MODEL WITH	HEAT			FPRINT			GBANK			SDT		
	Sz	P	M	Sz	P	M	Sz	P	M	Sz	P	M
HEAT	100	4	12⋆	100	4	12⋆	100	8	12⋆	100	4	4⋆
	200	12	12	200	8	12⋆	200	8	12⋆	200	8	6⋆
	300	12	12	300	12	12	300	12	12	300	10	8⋆
	400	12	12	400	12	12	400	12	12	400	10	10
	Sz	P	M	Sz	P	M	Sz	P	M	Sz	P	M
FPRINT	100	12	12	100	12	12	100	12	12	100	10	4⋆
	200	12	12	200	12	12	200	12	12	200	10	6⋆
	300	12	12	300	12	12	300	12	12	300	10	8⋆
	400	12	12	400	12	12	400	12	12	400	10	10
	Sz	P	M	Sz	P	M	Sz	P	M	Sz	P	M
GBANK	100	12	12	100	12	12	100	12	12	100	10	4⋆
	200	12	12	200	12	12	200	12	12	200	10	6⋆
	300	12	12	300	12	12	300	12	12	300	10	8⋆
	400	12	12	400	12	12	400	12	12	400	10	10
	Sz	P	M	Sz	P	M	Sz	P	M	Sz	P	M
SDT	100	4	12⋆	100	12	12	100	6	12⋆	100	4	4
	200	6	12⋆	200	12	12	200	10	12⋆	200	8	6⋆
	300	10	12⋆	300	12	12	300	12	12	300	8	8
	400	12	12	400	12	12	400	12	12	400	10	10

Sz=Size; P=predicted min. num. of processors;
M=measured min. num. of processors; ⋆=misprediction

$$\frac{76.226 - 80.873}{8} = -0.581$$

Note, the execution times at that size are about 75 s but the absolute error in execution time due to the misprediction is 4.647 s. Similar results can be established throughout the tables by looking at the actual run-time measurements.

Table 5.6: results of minimizing execution time for Fast Ethernet

APPLICATION PREDICTED

APPLICATION USED TO CALIBRATE MODEL WITH	HEAT			FPRINT			GBANK			SDT		
	Sz	P	M	Sz	P	M	Sz	P	M	Sz	P	M
HEAT	100	4	12⋆	100	4	12⋆	100	4	12⋆	100	2	6⋆
	200	12	12	200	8	12⋆	200	8	12⋆	200	8	8⋆
	300	12	12	300	12	12	300	12	12	300	10	10
	400	12	12	400	12	12	400	12	12	400	10	10
FPRINT	Sz	P	M	Sz	P	M	Sz	P	M	Sz	P	M
	100	12	12	100	12	12	100	12	12	100	10	6⋆
	200	12	12	200	12	12	200	12	12	200	10	8⋆
	300	12	12	300	12	12	300	12	12	300	10	10
	400	12	12	400	12	12	400	12	12	400	10	10
GBANK	Sz	P	M	Sz	P	M	Sz	P	M	Sz	P	M
	100	12	12	100	12	12	100	12	12	100	10	6⋆
	200	12	12	200	12	12	200	12	12	200	10	8⋆
	300	12	12	300	12	12	300	12	12	300	10	10
	400	12	12	400	12	12	400	12	12	400	10	10
SDT	Sz	P	M	Sz	P	M	Sz	P	M	Sz	P	M
	100	2	12⋆	100	12	12	100	6	12⋆	100	4	6⋆
	200	8	12⋆	200	12	12	200	12	12	200	8	8
	300	10	12⋆	300	12	12	300	12	12	300	8	10⋆
	400	12	12	400	12	12	400	12	12	400	10	10

Sz=Size; P=predicted min. num. of processors;
M=measured min. num. of processors; ⋆=misprediction

Table 5.7: results of minimizing execution time for Ethernet

		Ethernet of SPARC 10s				
		size	**min. by metric**	**min. by measure**	**slope (s/proc)**	**|err| (s)**
calibrated by HEAT	HEAT	100	12	10	$\frac{183.683-178.788}{2} = -2.447$	4.895
		200	12	12		
		300	12	12		
		400	12	12		
	FPRINT	100	12	10	$\frac{36.493-37.762}{2} = -0.6345$	1.269
		200	12	12		
		300	12	12		
		400	12	12		
	GBANK	100	12	12		
		200	12	12		
		300	12	12		
		400	12	12		
calibrated by FPRINT	HEAT	100	12	10	$\frac{183.683-178.788}{2} = -2.447$	4.895
		200	12	12		
		300	12	12		
		400	12	12		
	FPRINT	100	12	10		
		200	12	12		
		300	12	12		
		400	12	12		
	GBANK	100	12	12		
		200	12	12		
		300	12	12		
		400	12	12		
calibrated by GBANK	HEAT	100	12	10	$\frac{183.683-178.788}{2} = 2.447$	4.895
		200	12	12		
		300	12	12		
		400	12	12		
	FPRINT	100	12	10		
		200	12	12		
		300	12	12		
		400	12	12		
	GBANK	100	12	12		
		200	12	12		
		300	12	12		
		400	12	12		

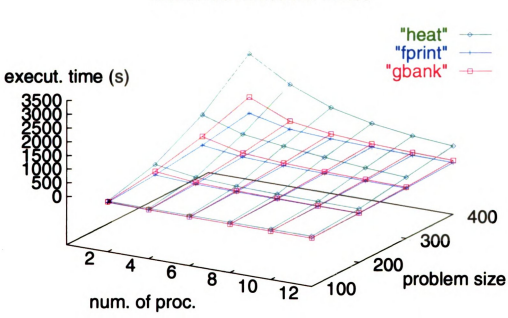

Figure 5.14: measured execution times of all three applications

5.6 Other Applications

Throughout this chapter, we discuss the use of the metric for one particular application. In this section, we present a number of applications that may take advantage of the features of the metric.

Our metric can be viewed as a metric describing the size-processor relationship. In this chapter, we discuss the specific application where the size of a given problem is known and we want to find the optimal number of processors. In other situations, it may be desirable, given the platform, to know the size of a problem that can be solved efficiently. Bigger problems can be solved, but an engineer may prefer to know, "what precision do we get before my system starts degrading in performance?"

In certain languages, the compiler has to make decisions about when to re-organize the data layout between different phases of the program. For example, the optimal distribution of data for one loop nest may not be optimal for the next loop nest. The compiler has to decide whether to redistribute the data between loop nests or execute one of the loop nests with a less-than-optimal organization of the data. A third possibility is for the compiler to find a layout that is optimal for the sequential combination of the loops, which may be sub-optimal for both nests individually but requires no redistribution. The problem is more complicated when several loop nests are involved. These questions can be addressed if the compiler has a strong predictor of performance. Even in HPF, where the user makes many of these decisions, the issue still arises when library routines are called.

Finally, as mentioned in the introduction, early frameworks used goals such as "maximum parallelism." Later, due to communication costs, researchers showed that it is impor-

tant to reduce or eliminate communication to make the parallelism worthwhile. Huang and Sadayappan [35] and Lim and Lam [46] have proposed algorithms to produce communication-free programs. The algorithms are driven by goals to reduce or eliminate communication. This approach assumes a fixed number of processors and the goal is to find parallel tasks that have no communication costs. An important contribution of our metric is that it includes problem size, the number of processors, and hardware characteristics that make striking a balance between the two extremes feasible. When necessary, an algorithm might choose to find a communication-free partition of the loop iterations that use a subset of the processors available. (A single processor is always communication-free.)

5.7 Summary

Our goal in this chapter is to show that a performance metric can be developed for a NoW that will determine the ideal number of processors needed to execute a program. A performance metric is challenging because of the large number of variations in a NoW that do not exist for dedicated distributed memory multiprocessors. We highlight the importance of observation skew. The work on observation skew and the statistical analysis was introduced in [60]. We showed that any system that models NoW will need to carefully measure the different components.

Our experimental results verify that the model can statistically describe the behaviors of our test applications. Alternate models were considered and rejected for statistical reasons. Finally, we performed a simple experiment to validate that this approach can be applied to performance prediction in NoWs. The validation results were mixed but nonetheless

several conclusions can be drawn. One is that applications were able to predict subsequent runs of the same application when their parameters were varied. This fairly limited result is the stated goal of some projects and is important for some settings that do run the same applications repeatedly. Another conclusion from the validation experiment is that certain applications are significantly worse at predicting other applications' behaviors. Identifying what makes a good application for calibrating this model is unanswered.

We believe this approach is an important step in compiling high performance applications for Networks of Workstations. By including run-time information, our metric can respond to an important feature of NoWs. Namely, that NoWs can take advantage of the latest off-the-shelf components to improve their performance characteristics.

Chapter 6

Conclusion

As HPCs continue to advance, their performance characteristics change and either the applications or the compilers need to change with them. Applications written for vector computers will not — without adjustment — immediately perform best on a NoW. But with advances in restructuring compiler frameworks, applications written previously can be efficiently executed on today's HPCs and applications written today can be efficiently executed on tomorrow's HPCs. Addressed in this work are two issues fundamental to strengthening and broadening this statement. We focus in Chapters 3 and 4 on increasing the number of loops that can be transformed by the restructuring compiler frameworks. In Chapter 5 we develop an advanced metric to drive compiler frameworks for NoWs. As NoWs are on the verge of becoming a mainstay HPC, the metric of Chapter 5 is especially important because it adapts to the performance characteristics of a NoW.

In this final chapter, we summarize this work, state its contributions, and discuss some future directions.

6.1 Summary

Imperfect Loop Nests The first issue addressed by this dissertation is the problem of extending matrix-based frameworks to imperfectly nested loops. We present two approaches.

The first starts with an analysis of loop nests in existing FORTRAN codes and then shows how to use transformations that are not specifically matrix-based to design a compiler with the same reordering abilities of a matrix-based restructuring compiler but which handles a larger class of loop nests (the apperfect class). Using the information gathered from an analysis of loops in engineering and scientific programs, we describe the design of a matrix-based restructuring compiler.

Our analysis of 641 loop nests revealed a number of interesting characteristics. First, of all the techniques developed to convert imperfect loop nests into perfect loop nests, only the composition of two (loop distribution and scalar forward substitution) are necessary. Integrating other techniques will not increase further the number of loop nests converted. Second, these two techniques handle a large number of the cases that occur in scientific code. Of the 31 loop nests that remained imperfect, after applying the transformation techniques discussed, most were still imperfect. This is due to limitations of the algorithms chosen by the users. (Thus either the compiler or the user must change the *user's* algorithm to execute the loop in parallel.) Some of the loops remained imperfect because of induction variables. It appears likely that future improvements in compiler transformations can be developed for this case. Another eight cases remained imperfect because of the nesting structure of the loop. Also, there were 32 loop nests that could not be analyzed because of

fundamental loop and if/then/else structure problems or the presence of GOTOs. Finally, 22 loops had I/O statements. Thus, we could not reorder their iteration space.

Using the information from the study, we discuss an optimal ordering of transformations for a restructuring compiler. Our order maximizes the number of perfect loop nests (based on known transformations) and lets the matrix-based transformation algorithms generate maximum parallelism. Loop nests that did not benefit from the matrix-based transformation procedure are restored to their original form. Thus, the code is never adversely affected by applying the transformation sequence.

The phase method is a novel, matrix-based approach that integrates the class of imperfectly nested loops and Banerjee's theory of unimodular transformations. It has the advantage of restructuring loops that other techniques cannot while maintaining the proofs of optimality in the original theory. The technique works by incorporating data dependence information from all levels in the loop nest, and then, after applying any standard transformation algorithm, the code generation part of the algorithm can be replaced by an algorithm that correctly outputs the transformed code in three phases. In this dissertation we apply the phase method to doubly-nested loops only.

Performance Metric The second matrix-based restructuring compiler issue addressed in the dissertation is the topic of Chapter 5. Namely, it is the development of a performance metric to guide a framework-based restructuring compiler. Performance metrics have been used for many years to implicitly or explicitly guide restructuring compilers. But for NoWs, many of these approaches are ill-suited because they require detailed information about the communication subsystem. We propose a system that uses instrumented applications to

gather information about the communication and computation performance of a specific NoW. This information is used to calibrate a model of the system which is then used as a performance metric for that NoW. If the NoW changes or a different NoW is considered, the metric is recalibrated. This ability to adapt is extremely important for NoWs since they rely heavily on commercial, off-the-shelf components that necessarily do vary. Furthermore, this is a complication that does not exist for traditional distributed memory multiprocessors.

Towards this goal, we establish several specific results in this paper. We document a process for instrumenting application codes and analyzing the resulting data. We formulate a generic model upon which to base a metric and show that such a model could be formulated with the necessary statistical properties needed to do the multiple linear regression in the calibration stage.

Crucial to modeling a NoW is an understanding that measurements made by individual workstations during the calibration stage may be skewed by workload imbalance and operating system delays. We show that a metric can be developed to model the performance of an application on a specific NoW if the measurements made by individual workstations are summarized correctly. We show that the metric can be calibrated such that the performance of an application on a specific NoW can be modeled.

In addition to checking that linear regression is statistically valid, we show that the model we develop accurately represents measured samples of a real application by a visual inspection of the graphs, an analysis of residuals, and by checking the proportion of total variability. We found that a model can be developed, using prior runs of an application, to predict the performance of new runs with different parameters (different problem size or different number of processors). We had moderate success using one application to predict

another application's performance. A key factor appears to be the choice of the application used to calibrate the model.

6.2 Future Directions

Although most of the imperfectly nested loops that appear in scientific codes have been addressed, there are still some important loop nests, such as the Cholesky decomposition routine that are imperfect and not easily handled by any compiler transformation. Other than a few special cases, though, most of the problems associated with imperfect loop nests have been resolved.

NoWs have been discussed and used for over five years, but much remains unknown about these platforms — especially their general suitability as HPCs. For example, all of our experiments were conducted on platforms that allowed users to log in while our applications ran. Our approach was to monitor the systems closely and keep a log of pertinent aspects of the machine state. If a user disrupted one of our experiments by using one of the machines, an e-mail message informed us to remove that specific record from the samples. One case "slipped in" to the records. The user logged into one workstation and was idle as defined by the usual system measures (the Unix load command showed that the load was less than 0.02 every time it was measured). This occurrence affected several executions of all three applications. Despite appearing to do no CPU-intensive activity, the user affected every application that was executed while the user was logged into the system. The executions showed up as extreme outliers in the residuals — sometimes 2–8 times larger in magnitude than the next biggest residual. This is anecdotal evidence that

NoW systems are extremely sensitive to sharing the CPU or memory on a workstation with another user.

We also found, by accident, that NoWs are susceptible to network contention. For the Ethernet network of SPARC 10s platform, we had access to 15 workstations. The network had an additional machine that was used to capture frames of video from a security camera and transmit them to a server. This output was every 6 seconds. When we ran our experiments originally we did not take note of the camera and its presence on the network. What resulted was some distributions that were bimodal — distinctly not normal. The graph presented Figure 6.1 is a striking example of the effects of the security camera. The first graph shows when the camera was operating and the second when it was not. The first is clearly not normally distributed. This is more anecdotal evidence and suggests that NoWs are especially susceptible to network interference.

A third area that has not received any attention, despite its large potential, is the execution of more than one process per a workstation. The Unix load, sampled several times during the execution of one our test applications indicated that the processor was way ahead of the network. Instead of being close to 1.0 (always computing), the average load was approximately 0.6. As microprocessors continue to improve performance faster than networks and workstations are configured with multiple processors, the microprocessor will spend even more of its time idle. One solution (which as just begun to be explored) is to put more than one process on a workstation [64]. This uses time-sharing on each node to use the hardware more effectively and presents a very new modeling problem.

Thus subtle interactions that may occur because NoWs are rooted in general-purpose, individual machines, do have an obvious impact on performance. In light of these last

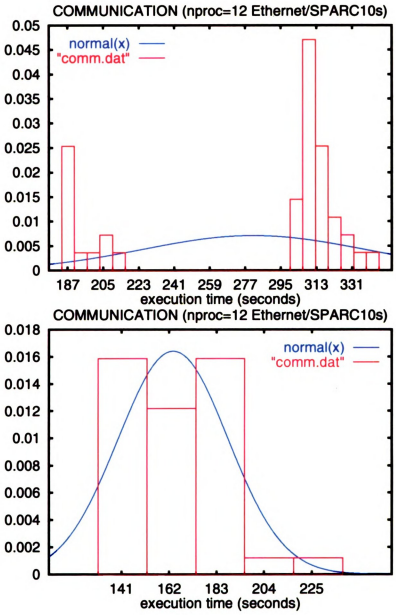

Figure 6.1: with and without the camera operating

three examples, we believe there is a great deal more to be discovered about Networks of Workstations. By improving the state of High Performance Computing, we hope to add to the number of tools available to engineers and scientists for performing computations.

APPENDICES

Appendix A

Restructuring Compilers

In this appendix, we give a brief summary of our experiences compiling FORTRAN programs and using a number of compiler toolkits. We anticipate that this information will prove helpful to other researchers pursuing common interests. In the first section, we discuss three problems we had to overcome using FORTRAN as a source language. In the second section we discuss specific toolkits.

A.1 Fortran

In the course of carrying out this research, we discovered three practical problems while attempting to parse FORTRAN programs. Because FORTRAN was originally designed without the idea of tokens and whitespace delimiters, the language is difficult to parse with modern compiler tools. FORTRAN has changed substantially over the years and some very archaic constructs are still used occasionally. Thus, compilers have to handle each construct as

a special case. Also, because the GOTO statement is a prominent feature in FORTRAN, programs are frequently unstructured.

Because FORTRAN does not always consider whitespace significant, parsing FORTRAN is more difficult than other common languages. Since our research concentrates on problems in a later stage of the compiling process, we searched for tools to handle this step for us. At the time, the only toolkit publicly available was Sage. Since then, other toolkits have emerged (see the next section). Although Sage was relatively new and untested, we considered Sage a better choice than developing our own. This choice caused trouble initially because Sage was an attempt to merge two independent projects that had a common ancestor (SIGMACs). The documentation was weak and simply wrong in places (i.e., we frequently used the source code to find the true specification for procedure calls in their library.) One of the two independent projects was the Sigma Toolkit, version 0.2beta. The version number suggests that the authors did not have a lot of confidence in the toolkit. Nevertheless, we found the Sigma Toolkit the most stable portion of the Sage toolkit. In the final process, we eliminated all references to the Sage toolkit except for the Sigma toolkit portion.

Although we found that the parser always parsed FORTRAN correctly, there were a number of problems in the toolkit's transformation library. Because the the documentation listed subroutines to do Scalar Forward Substitution and Loop Distribution, we thought we would be able to use their routines and complete our project quickly. Unfortunately, after we started collecting and analyzing data, we discovered that the toolkit's definitions for these transformations were significantly more conservative than expected. For example, Loop Distribution did not work (correctly) for imperfect nests — exactly our intended

use! To complete our project we had to substitute our own routines for both of these transformations.

Even with this additional work, it was less effort utilizing the toolkit than writing our own FORTRAN parser and transformation library.

A second difficulty we had to overcome with FORTRAN was the large variety of statements. Because of its long history, FORTRAN has a number of different ways to accomplish the same task. A DO loop may use a line number, line number with a CONTINUE statement, or a DO/ENDDO pair to indicate what code is to be repeated. IF statements have various forms. There are computed GOTOs and other rare statements that still occur in some programs. We dealt with this variety of formats by running some conversion steps first. Thus, all DO loops were converted to have an ENDDO form. Most IF statements that had a simple IF/ENDIF structure but were written with a GOTO were converted to have an ENDIF. Thus, the input to our analysis procedures was fairly uniform. There were specific instances (such as computed GOTOs) where we had to simply ignore the loop nest and not perform any transformations.

Finally, in most modern languages, GOTO statements are either non-existent or their use is discouraged. For a long time, the GOTO statement in FORTRAN was not only encouraged but required to make other structures such as WHILE loops. The result is that the occurrence of unstructured code is much more common in FORTRAN than in other languages. This is similar to the previous problem in that unstructured code prevents our transformations.

A.2 Toolkits

In this section we discuss some of the compiler toolkits and libraries that are available presently.

The Parafrase II source-to-source compiler is available commercially. Unfortunately, we received C source code that was difficult to work with and virtually impossible to revise. (All of the comments were removed and the macros expanded.) Although it is clear that the design of Parafrase II makes it easy to incorporate new transformations, it is also clear the intent is not for purchasers to add new transformations. As a restructuring compiler, the product performed as advertised but we could not use it as a toolkit for compiler research because the source code could not be revised easily.

TINY is another program that was considered. It is a simple program with its own source language. It was intended to be a learning tool for for students. As such, it is worthwhile. We recommend it as starting point but agree with its creator, M. J. Wolfe, that it is not a substitute for a full dependence analyzer and compiler front-end [71].

Nevertheless, the researchers at the University of Maryland did use TINY as the starting point for a research project. They incorporated their Omega dependence test, added features to the interface, and a generate-and-test search for transformations. It still uses TINY's original source language as opposed to FORTRAN so we find that, at best, it can only be used for small test programs. We believe that the simplicity of TINY was a strength. Omega's value is that it demonstrates the Omega test.

Sage++ has sprouted from the origins of the SIGMACs, Sigma Toolkit, and Sage projects. Whereas the Sage library used a Lisp-like list form for passing arguments, Sage++

offers an object-oriented interface. It is released but is still relatively new. A number of bugs pertaining to Sage++ have been reported. For example, with Sage, one can write a program that processes multiple source (FORTRAN) files. Soon after it was released Gannon, the principle investigator of the Sage++ Project, recommended that only a single file be parsed by Sage++ until it becomes more stable.

A number of papers from Stanford have referred to their SUIF compiler. Although the papers are dated back to 1992, the compiler was released to the public after the work in Chapter 3 was completed in 1994. Although we have not used it, we suspect — based on how long it has been used internally and the reputation of the source — that it is a solid compiler. In addition, its stated purpose is to be a compiler testbed for new research on transformations. It is likely that our future work will use this compiler.

Appendix B

Data-Parallel System and Applications

In this appendix, we give a complete description of our data-parallel system and the test applications. The primary differences between the presentation here and the concise version in the main text are the details for coding the specific serial-to-data-parallel transformations.

B.1 Data-Parallel System

To implement the applications (described next) on a NoW, we developed a simple C++ data-parallel library. The library has classes to handle distributed arrays, spawning and managing of SPMD processes, group/neighbor communications, and timing information. Our system uses static scheduling. The parallelism is determined by the physical location of data; namely, it uses the owner-computes rule.

We use PVM 3.3.11 as a basis for the library. PVM is called directly for messages, thus there is no overhead, but we use our library (`spmd.left()` and `spmd.right()`,

for example) to get neighbor task IDs in a linear topology, which PVM does not support.[1] While all of our experiments are based on PVM, our system is not tightly bound to PVM; other message-passing systems could easily be substituted. Also, tasks are created such that there is always one process per workstation in the NoW.

The distributed arrays are built with a `plane` class that allows general memory layouts for 2D arrays, including distribution of an array across workstations. The class has built-in support for "overlap" regions which are used by data-parallel compiler transformations that combine several messages and transfer them together. For example, a whole column may be moved from one process to the overlap region of another process prior to executing a loop nest. This is called message coalescing and is an important transformation for efficiency [34]. There is no longer a need for communication within the loop nest because the remote references are now available in the overlap region.

Our library facilitates the instrumentation and calibration of the programs with a timing class. Details were given in 5.2.

All of the applications were compiled "by hand" because, in order to make the necessary changes automatic, we would require access to the data parallel compiler source, which we did not have. Our guiding principle in the translation of the application to a data-parallel SPMD program was to emulate the behavior of current compilers. We applied optimizations that have been well-documented and shown to be feasible, such as coalescing of messages, but we resisted changes that humans recognize but compilers would not.

[1] PVM does support group operations but it does so by creating an extra task. This extra task led to efficiency problems for us, so we abandoned it.

Four major structures were converted in the serial to data-parallel process. We describe each conversion below. In all the examples, the method `get(i,j)` accesses one element in a distributed array where the location of the i^{th} elements are controlled by the distribution object `d0`. (In our library, `d0` is declared with the size of the dimension and a type of distribution, `BLOCK`, for example.)

- Message-Passing for Remote References

  For any loop nest that references distributed memory, we considered two ways of converting the loop. If the data dependences are uniform, the loop limits can be restricted to just the iterations owned by the process. Otherwise, the loop cycles through all of the iterations and the loop body is guarded by a conditional that is true when the process "owns" that iteration. Figure B.1 (a) is a serial loop nest and (b) and (c) show the two parallel solutions we use.

- Input/Output

  For scatter operations (where data are input on one workstation and then distributed to their owners), we replace the actual I/O statement with the lines of pseudocode shown in Figure B.2. Gather operations (data are transmitted from the owners to the I/O workstation and then outputted) are handled in a similar fashion.

- Global Reduction

  We implement a reduction in the most straight-forward way. The master does $n-1$ receives and performs all of the operations. Then, the master distributes the result in $n-1$ sends to each workstation. The workstations, other than the master, do a send followed by a receive.

Serial

```
for( i=0 ; i<(n-m) ; i++ )
    ··· = xmat.get(i,j)
```

(a)

Parallel 1

```
for( i=d0.maplb(0) ; i<d0.mapub(n-m) ; i++ )
    ··· = xmat.get(i,j)
```

(b)

Parallel 2

```
for( i=0 ; i<(n-m) ; i++ )
    if( d0.iown(i) )
        ··· = xmat.get(i,j)
```

(c)

Figure B.1: changing serial loops to data-parallel loops

- Barrier Synchronization

  We implement a barrier synchronization as a global reduction with no operation. That is, the master does $n-1$ receives followed by $n-1$ sends.

All of the techniques that were performed "by hand" can be implemented in a compiler. Based on the results presented later, it is profitable to add these translations to a compiler. But, prior to our results, the considerable effort of making a compiler would have been risky.

```
If I am the I/O node
    Do the I/O  statement
    If I own the data
        copy into place
    Else
        calculate destination
        send
    Endif
Else
    If I own the data
        receive
    Endif
Endif
```

Figure B.2: pseudocode for scatter/gather operations

It was our intention to make the library unobtrusive. As mentioned, our library adds no overhead to the sending of messages because the PVM calls are directly inserted. The library adds one collective communication and it appears at the end. It is not a significant factor in the overall execution time.

B.2 Heat Transfer Application

The first application calculates a sequence of images that indicate the transient heat transfer through materials with different thermal coefficients. We assume a closed system, so it is unnecessary to model the heat capacities of the materials. In our simulated environment, we use a copper disk, surrounded by air with a circular heat source underneath as an input. The copper and heat source are shown in Figure B.3(a). Figure B.3 (b) and (c) are two frames from a typical sequence. Each pixel in the two output frames represents the temperature

Figure B.3: simulated environment (leftmost frame) and two video frames (middle and rightmost frames)

at that point by its intensity. Thus, an engineer can observe the heat transfer through the design. Intermediate frames are written to disk. Unlike many video applications, this application does not have frame-level parallelism because frame $(i-1)$ is needed to calculate the i^{th} frame. So, instead of calculating the frames in parallel, we distribute the columns of the array representing the image (and other associated arrays) across the processors. Since these slices are not independent, a data-parallel compiler, such as FORTRAN D or HPF, will generate the necessary communication, which we manually insert. A higher level compiler with automatic data decomposition would likely produce similar communication patterns. This application includes a number of different loop nests involving multiple distributed arrays, I/O procedures that gather individual frames to be written to disk, messages to move columns of data between neighbors, and global reductions.

B.3 Fingerprint Matching Application

The second application is the last stage in automatic fingerprint identification. In previous stages of the identification process, fingerprint features are extracted from digitized images and stored in a database. The features of an unidentified fingerprint are used to search the database for a match. Since two digitized images of the same fingerprint are unlikely to have precisely the same features, a "scoring" function determines how closely the features match. We presume that a typical system would test several unidentified fingerprints and would take advantage of the aggregate memory of a NoW to distribute the database. Thus, a single fingerprint is processed in our application by sending the features to each workstation where the search is done in parallel. After all of the local scores have been calculated, a parallel sort involving all of the processors sorts the scores so that the top ten scores (most likely matches) are on the I/O node where they are output for final verification (selection of the best match by a human).

This application has two main phases, each with different communication patterns and loop bodies. The bulk of the I/O is at the beginning when the application is loading the database and then occasional I/O as a new unidentified fingerprint is distributed.

B.4 Texture Segmentation Application

At another stage in the automatic fingerprint identification process, features are extracted from a digitized image. One technique uses a bank of Gabor filters to extract the features. In the spatial domain, this involves a series of convolutions with fairly large convolution masks (we use 32×32). Each mask is generated from a complex sinusoid and is effectively

a band-pass filter. For our tests, we used a grayscale "rainbow" image of various sizes and varied the filter orientation. We used three orientations, although larger banks of filters are frequently used. Our application reads in the original image and distributes the columns to the workstations. Every workstation calculates the current mask in parallel and the algorithm proceeds with the convolution over the image. Each filter produces a separate image so the original is not destroyed. The final step is to transmit the resulting images back to the I/O for output.

Besides feature extraction, Gabor filters are also used in other applications such as texture segmentation.

B.5 Spatial Decomposition Technique (SDT)

Analysis of electrically large objects, such as antennæ of various geometries, is a computationally difficult problem. A straight-forward formulation of the problem leads to densely-filled, numerically-intensive, complex-valued matrices. So it is not surprising that many techniques have been developed over the years to address this problem in formulations that are less computationally expensive. One of them is the SDT.

The SDT allows the total operation count to be reduced by introducing subobjects which are simpler to compute. We chose this approach because it is easily written in a data-parallel style and the parallel version requires both intensive computation and communication.

BIBLIOGRAPHY

Bibliography

[1] W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers*. PhD thesis, University of Illinois at Urbana-Champaign, November 1978.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[3] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4), October 1987.

[4] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *Conference on Programming Language Design and Implementation*, pages 126–138, 1993.

[5] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.

[6] Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Conference on Programming Language Design and Implementation*, pages 112–125, 1993.

[7] Eduard Ayguadé and Jordi Torres. Partitioning the statement per iteration space using non-singular matrices. In *ACM International Conference on Supercomputing*, pages 407–415, 1993.

[8] Utpal Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic, Boston, 1988.

[9] Utpal Banerjee. Unimodular transformations of double loops. In Alexandru Nicolau, David Gelernter, Thomas Gross, and David Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 192–219. Pitman Publishing, 1991.

[10] Utpal Banerjee, July 1992. Personal communication to R. Sass during a short course on Restructuring Compilers in Trento, Italy.

[11] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Norwell, Massachusetts, 1993.

[12] Utpal Banerjee. *Loop Parallelization*. Kluwer Academic Publishers, Boston, 1994.

[13] Vasanth Blalsundraram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A static performance estimator to guide data parallel partitioning decisions. In *Third ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 21–24, 1991.

[14] F. Bodin, S. Lelait, and D. Windheiser. Sigma toolbox. Technical Report Sigma Toolbox Manual, version 0.2alpha, Irisa and Indiana University, July 1993.

[15] François Bodin, Daniel Windheiser, William Jalby, Daya Atapattu, Mannho Lee, and Dennis Gannon. Performance evaluation and prediction for parallel algorithms on the BBN GP1000. In *Proceedings of 1990 International Conference on Supercomputing*, 1990.

[16] Mark Brehob, Travis Doom, Richard Enbody, William H. Moore, Sherry Q. Moore, Ron Sass, and Charles Severance. Beyond RISC - the Post-RISC architecture. Technical Report MSU-CPS-96-11, Department of Computer Science, Michigan State University, East Lansing, Michigan, 48824, 1996.

[17] Edmund Burke. *Reflections on the Revolution in France*. Oxford University Press, 1790.

[18] Soumen Chakrabarti, Manish Gupta, and Jong-Deok Choi. Global communication analysis and optimization. In *Conference on Programming Language Design and Implementation*, pages 68–78, 1996.

[19] David Chesney. *Matrix-Based Representations of Loop Transformations*. PhD thesis, Michigan State University, December 1995.

[20] Michal Cierniak and Wei Li. Unifying data and control transformations. In *Conference on Programming Language Design and Implementation*, 1995.

[21] Erik H. D'Holander. Partitioning and labeling of loops by unimodular transformations. *IEEE Transactions on Parallel and Distributed Systems*, 3(4), July 1992.

[22] Anne Dierstein, Roman Hayer, and Thomas Rauber. The ADDAP system on the iPSC/860: Automatic data distribution and parallelization. *Journal of Parallel and Distributed Computing*, 32(1):1–10, 1996.

[23] J.J. Dongarra and E. Grosse. Distribution of mathematical software via electronic mail. *Communications of the ACM*, 30, 1987.

[24] R. Eigenmann, J. Hoeflinger, G. Jaxon, and D. Padua. Cedar fortran and its restructuring compiler. In Alexandru Nicolau, David Gelernter, Thomas Gross, and David Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 1–23. Pitman Publishing, 1991.

[25] Thomas Fahringer. Using the P3T to guide parallelization and optimization effort under the Vienna Fortran Compilation System. In *IEEE Procedings of the Scalable High Performance Computing Conference*, 1994.

[26] Thomas Fahringer and Hans P. Zima. A static parameter based peformance prediction tool for parallel programs. In *Proceedings of International Conference on Supercomputing 1993*, pages 207–219, 1993.

[27] Richard Feynman. *Surely You're Joking, Mr. Feynman!* W. W. Norton & Company, 1986.

[28] K. Gallivan, D. Gannon, W. Jalby, A. Malony, and H. Wijshoff. Behavioral characterization of multiprocessor memory subsystems: A case study. In *ACM Sigmetrics Performance Evaluation Review*, volume 17, pages 79–88, 1989.

[29] Dennis Gannon, Jenq Kuen Lee, Bruce Shei, Sekhar Sarukaiand Srivinas Narayana, Neelakantan Sundaresan, Daya Atapattu, and Francois Bodin. Sigma II: A toolkit for building parallelizing compilers and performance analysis systems. In *Proceedings of the Programmming Environments for Parallel Computing*, Edinburgh, Scotland, April 1992.

[30] David A. Garza-Salazar and Wim Böhm. Reducing communication by honoring multiple alignments. In *International Conference on Supercomputing*, pages 87–96, July 1995.

[31] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In *Conference on Programming Language Design and Implementation*, pages 15–29, 1991.

[32] Manish Gupta and Edith Schonberg. Static analysis to reduce synchronization costs in data-parallel programs. In *Principles of Programming Languages*, pages 322–332, January 1996.

[33] David Alejandro Padua Haiek. *Multiprocessors: Discussion of Some Theoretical and Practical Problems*. PhD thesis, University of Illinois at Urbana-Champaign, October 1979.

[34] Seema Hiranandani, Ken Kennedy, , and Chau-Wen Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *International Conference on Supercomputing*, July 1992.

[35] C.-H. Huang and P. Sadayappan. Communication-free hyperplane partitioning of nested loops. *Journal of Parallel and Distributed Computing*, 19(1):90–102, 1993.

[36] François Irigoin and R. Triolet. Supernode partitioning. In *Fifteenth Annual ACM SIGACT-SIGPLAN Symposium of Programming Languages*, pages 319–329, January 1988.

[37] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-2995.1, University of Maryland, College Park, April 1993.

[38] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in the ParaScope Editor. In *International Conference on Supercomputing*, June 1991.

[39] K. Kennedy, K. S. McKinley, and C. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2:329–341, July 1991.

[40] Ken Kennedy and Ulrich Kremer. Automatic data layout for distributed-memory machines. *Transactions on Programming Languages and Systems*, 20(4), July 1998.

[41] David J. Kuck. *The Structure of Computers and Computations*. Wiley, New York, 1978.

[42] David J. Kuck. *High Performance Computing: Challenges for Future Systems*. Oxford University Press, New York, 1996.

[43] Leslie Lamport. The parallel execution of do loops. *Communications of the ACM*, 17(2), February 1974.

[44] Wei Li and Keshav Pingali. Access normalization: Loop restructuring for numa compilers. In *Proceedings of Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 285–295, Boston, Massachusetts, October 1992. Also appeared in SIGPLAN Notices 27(9).

[45] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. Technical Report TR 92-1294, Department of Computer Science, Cornell University, Ithaca, New York 14853, June 1992.

[46] Amy Lim and Monica S. Lam. Communication-free parallelization via affine transformations. In *Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[47] Lee-Chung Lu. A unified framework for systematic loop transformations. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 28–38, April 1991.

[48] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of International Conference on Supercomputing*, pages 85–97, June 1997.

[49] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Practical dependence testing. In *Conference on Programming Language Design and Implementation*, pages 1–14, 1991.

[50] Pankaj Mehra, Catherine H. Schulbach, and Jerry C. Yan. Comparison of two model-based performance-prediction techniques for message-passing parallel programs. In *ACM SIGMETRICS Performance Evaluation Review*, pages 181–190, May 1994.

[51] U. Meier and R. Eigenmann. Parallelization and performance of conjugate gradient algorithm on the cedar hierarchical-memory multiprocessor. Technical Report 1035, University of Illinois at Urbana-Champaign, Center for Supercomputing R&D, 1990.

[52] David Padua, David Kuck, and Duncan Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, C-29(9), September 1980.

[53] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman Publishers, San Mateo, California, 1990.

[54] J.-K. Peir and Ron Cytron. Minimum distance: A method for partitioning recurrences for multiprocessors. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 217–225, 1987.

[55] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten. The structure of Parafrase-2: An advanced parallelizing compiler for C and Fortran. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Fourth Workshop on Languages and Compilers for Parallel Computing*. MIT Press, 1990.

[56] William Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8), August 1992.

[57] V. Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. *IBM Journal of Research and Development*, 35(5/6), September 1991.

[58] Ron Sass and Matt Mutka. Enabling unimodular transformations. In *Proceedings of Supercomputing '94*, pages 753–762, November 1994.

[59] Ron Sass and Matt Mutka. Transformations on doubly nested loops. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 343–346, August 1994.

[60] Ron Sass and Matt Mutka. Statistical behavior of cluster computing applications. In *Conference on Cluster Computing '97*, March 1997.

[61] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[62] Weijia Shang and Jose A. B. Fortes. Independent partitioning of algorithms with uniform dependencies. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 26–33, 1988.

[63] Bruce Shei and Dennis Gannon. Sigmacs: a programmable programming environment. In Alexandru Nicolau, David Gelernter, Thomas Gross, and David Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 88–108. Pitman Publishing, 1991.

[64] Betty H.C. Cheng Stephen W. Turner, Lionel M. Ni. Time and/or space sharing in a workstation cluster environment. In *Proceedings of Supercomputing '94*, pages 630–639, November 1994.

[65] H. G. Wells. *The Shape of Things to Come.* The McMillan Company, New York, 1933. The title of a book by H. G. Wells.

[66] Debbie Whitfield and Mary Lou Soffa. An approach to ordering optimizing transformations. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 137–146, March 1990.

[67] Michael E. Wolf. *Improving Locality and Parallelism in Nested Loops.* PhD thesis, Stanford University, August 1992.

[68] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4), October 1991.

[69] Michael E. Wolf and Monica S. Lam. Maximizing parallelism via loop transformations. In Alexandru Nicolau, David Gelernter, Thomas Gross, and David Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 243–259. Pitman Publishing, 1991.

[70] Michael Wolfe. Scalar vs. parallel optimizations. Technical Report CS/E 90–010, Oregon Graduate Institute of Science and Technology, 19600 NW von Neumann Drive, Beaverton, OR 97006, 1990.

[71] Michael Wolfe. The tiny restructuring research tool. In *Proceedings of 1991 International Conference on Parallel Processing*, St. Charles, Illinois, 1991.

[72] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman Publishing, 128 Long Acre, London WC2E 9AN, 1989.

[73] M.J. Wolfe. Techniques for improving the inherent parallelism in programs. Technical Report 78-929, Department of Computer Science, University of Illinois at Urbana-Champaign, July 1990.

[74] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, New York, 1991.