COLLECTIVE ANALYSIS AND TRANSFORMATION OF LOOP CLUSTERS

by
Russell Olsen

School of Computer Science
McGill University, Montreal

May 1992

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

Abstract

Hardware alone is insufficient to allow high-performance computer systems to become successful; sophisticated compilers are also required. Compilers for systems of this type will achieve their effectiveness, not from any single optimizing transformation, but from advanced program analysis and selective application of many proven code-improving transformations. One such analysis technique which holds promise, especially for FORTRAN-90 applications, is a *collective analysis* technique developed by Sarkar and Gao which uses graph coloring to determine whether a cluster of loops can be processed in a pipelined manner [SG91]. The loop clusters we consider in this case are those consisting of loops which define arrays and successor loops which use them. When array elements which are generated in one loop are able to be referenced in the same order by successor loops, the loops are *compatible* for pipelined processing. On a multiprocessor, pipelined processing is accomplished by running the constituent loops on different processors; whereas on a uniprocessor, the loops are fused and intermediate arrays eliminated through a process of *collective transformation*, itself a process which often creates additional opportunities for code improvement.

In this document we describe enhancements to Sarkar and Gao's loop coloring technique which improve transformation effectiveness of optimizing compilers for uniprocessors; we also describe a minimum-cost partitioning heuristic to achieve efficient transformation in the event a loop cluster contains non-compatible loops. Based upon empirical evidence, we then show the benefit from loop transformation can be substantial, in certain instances resulting in four-fold speedup for certain types of codes. As part of our experiments, we use sample codes for both compatible and non-compatible loop clusters, run on each of three different types high-performance uniprocessors: a RISC workstation, a superscalar workstation, and a mainframe vector processor. Processor and cache simulations are then used to further substantiate our results. Overall, these experiments illustrate the advantages of using collective analysis and transformation within optimizing compilers for each type of machine.

Résumé

De nos jours, un support materiel efficace n'est pas suffisant pour assurer qu'un ordinateur haute-performance ait du succès; l'utilisation de compilateurs sophistiques est aussi nécessaire. Les compilateurs pour ce type de machine dérivent leur efficacité, non pas d'une seule technique d'optimisation, mais bien de l utilisation d'un ensemble d'analyses et de transformations. Une approche prometteuse, particulièrement pour les applications écrites en FORTRAN-90, est l'*analyse collective* développee par Sarkar et Gao, approche qui utilise une technique de coloration de graphes dans le but de déterminer si un ensemble de boucles peut être traité à l'aide d'un pipeline [SG91] Les ensembles de boucles que nous allons étudier sont les boucles servant a definir des tableaux de même que celles utilisant les tableaux ainsi définis Lorsque les elements d'un tableau produits par une boucle sont utilisés, par une autre boucle, dans le même ordre que celui où ils ont été produits, on dit alors que les deux boucles sont *compatibles* vis-à-vis un traitement par pipeline Dans une machine a plusieurs processeurs, un traitement par pipeline peut être réalisé en exécutant les diverses boucles sur des processeurs différents; par contre, sur une machine à un seul processeur les boucles peuvent être fusionnées et les tableaux intermédiaires peuvent être éliminés grâce a un processus de *transformation collective*, lequel processus peut lui même créer de nouvelles possibilités d'optimisation du code.

Dans ce mémoire, nous décrivons certaines améliorations à la technique de coloration de graphes proposée par Sarkar et Gao dans le but d'améliorer son efficacité pour des machines à un seul processeur, nous décrivons aussi une heuristique de décomposition à coût minimum dans le cas d'un ensemble de boucles non compatibles A l'aide de résultats empiriques, nous montrons ensuite que les avantages d'une telle transformation peuvent être substantiels, conduisant, dans certains cas a des programmes quatre fois plus efficaces Dans le cadre de nos expériences nous utilisons des programmes contenant des boucles compatibles ainsi que des boucles non compatibles, chaque programme étant exécuté sur trois types de machines a haute performance: une station de travail RISC, une station de travail super-scalaire et un processeur vectoriel. Des simulations de processeurs et de mémoires caches sont aussi utilisées pour confirmer nos résultats. De façon générale, ces diverses expériences illustrent les avantages d'utiliser les techniques d'analyse et de tranformation collective.

Acknowledgements

Few people are able to successfully complete the research required for a Masters thesis without some type of support and commitment from family, advisors, and friends. I am certainly no exception for along the way there were many people who helped me and to whom I owe so much.

First and foremost I gratefully acknowledge the enormous personal sacrifice and unflinching support of my wife Yoshiko. Only those who have actually been around her these past few years could possibly appreciate the countless ways she has improved my life. For cheerfully enduring our many daily hardships and challenges, she is truly deserving herself of honorary degrees, cum laude, in creative financing, survival living, morale boosting, and motivational counseling. She has been my dearest and closest friend, not just during this past three years, but for all of our many splendid years together, and dearest closest friends I hope we always shall remain.

Secondly, I express my deepest appreciation to Prof. Guang Gao. His enduring patience and constant support were major factors keeping me going throughout this arduous ordeal. Foremost he provided me with the incentive I needed to study high performance computer architectures by introducing each new subject in a comprehensible and intellectually stimulating way, through his numerous interesting seminars and many challenging computer architecture courses (CS505, CS605 and CS762). As a result of his careful guidance and continued support throughout these past three years I was gradually, but ultimately, able to acquire both the confidence and expertise required to become a computer science professional, having little or no exposure to this fascinating but complex subject prior to coming to McGill. Prof. Gao is an extraordinary scientist and scholar, a person of enormous ability, vast knowledge and rare intuition, and he will always have my highest admiration and deepest respect.

I also feel a tremendous debt of gratitude to our very special friends who unhesitantly provided financial assistance to Yoshiko and me when we so desperately needed it. Nancy Henry, Jack and Nicole Lewis, and Malcom and Jean Short. Their unwavering trust and unbounded generosity will always be remembered and appreciated. Likewise I gratefully acknowledge Prof. Gao's financial support, as well as the financial support of NSERC and BNR.

Next I wish to gratefully acknowledge the contribution made by fellow classmates and members of the Advanced Computers and Program Structures (ACAPS) group at McGill: Erik Altman, Herbert Hum, Jean-Marc Monti, Chandrika Mukerji, Qi Ning, Ravi Shanker, Nematollaah Shiri, V.C. Sreedhar, Kevin Theobald, Guy Tremblay, Phillip Wong, and Bob Yates. Their association added immeasurably to the quality of my education both with respect to my understanding of the subject matter and to my enjoyment in being here. Through their many helpful discussions and our many pleasant gatherings together we have forged friendships which I hope will continue for many years to come. In this same respect, I express sincere gratitude to Shiela Sundaram and Vigay Grey. Their constant encouragement and support throughout this experience was also much appreciated.

I also gratefully acknowledge the significant contributions made by Dr. Vivek Sarkar at IBM Palo Alto Scientific Center and Radhika Thekkath at University of Washington, since the influence of their research is reflected throughout this document. Likewise, the assistance provided by Erik Altman and Chandrika Mukerji, with the instruction scheduler used during loop-transformation testing, was absolutely vital and deeply appreciated, especially during the hectic latter days of this research when both of these persons were so very busy with priority undertakings of their own.

Last, but certainly not least, I would be remiss were I not to recognize the substantial contribution of my parents, my daughter Danna, Yoshiko's family, and our many other friends who in their own special individual ways provided the additional encouragement and support necessary to make our stay in Montreal the satisfying and enjoyable experience that it was. Through their correspondence, their telephone calls, and their visits we have grown to appreciate and cherish each of these priceless relationships even more.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In spite of their great speed, supercomputers today are still not fast enough to solve many of the problems users would like them to be able to solve. As a result, machines of ever increasing computing power are needed, and although more advanced computers are constantly being developed, the introduction of a new machine usually comes at a cost of increased complexity. The reason for this complexity is simple: either machines be made to operate faster or they be made to do more work concurrently. Although clock speeds have been increasing, the potential for further clock-speed increase is limited because of a physical constraint on the rate at which VLSI, and more recently ULSI, logic is able to operate. On the other hand, concurrent processing involves overheads, such as communication and synchronization, which quickly erode the performance benefit which might otherwise accrue from concurrent hardware, especially if the hardware is not effectively managed.

As a consequence, hardware alone is insufficient to allow high-performance computer systems to achieve their full potential; some form of software support is essential, and because of the complexity inherent in controlling program execution on parallel hardware, this software support will necessarily take the the form of a highly sophisticated optimizing compiler. Compilers of this type will achieve their effectiveness, not from any single optimizing transformation, but from advanced program analysis and selective application of many proven code-improving transformations, likely requiring much closer programmer interaction than has been required of optimizing compilers in the past.

1.1 Scope of Research

The subject of this thesis is a new loop analysis and transformation technique which brings compiler technology closer to meeting the needs of newer high performance computer architectures. In specific terms, the technique provides a way of analyzing loop clusters in which the lead loops define array elements and successor loops use these elements to define yet other arrays. Array computations of this type could easily comprise a large portion of any scientific computation, especially if the application were written in a language such as FORTRAN 90. Consequently, being able to process arrays of this type efficiently is of fundamental importance. When array elements which are generated in one loop are able to be referenced in the same order by successor loops, the loops are *compatible* for pipelined processing. On a multiprocessor, pipelined processing is accomplished by running the constituent loops on different processors, whereas on a uniprocessor, the loops are fused and intermediate arrays eliminated through a process of *collective transformation*, a process itself which often creates opportunities for additional code improvement.

By considering clusters as a collection, rather than individually, a compiler is able to identify opportunities for transformation which would not be apparent were the loops considered individually, as in traditional loop optimization. As we show in this document, the benefit from applying a global perspective to loop transformation can be substantial, especially in compiling programs, such as FORTRAN-90 programs, which allow statements in vector notation.

The particular loop analysis methods we evaluate are based upon techniques developed by Sarkar and Gao [SG91]. Gao first raised the issue of collective loop analysis in the context of an *intra-block pipelining* problem, his primary focus being the development of compiler techniques for static dataflow architectures. Later, Sarkar and Gao developed strategies for solving several common subclasses of the problem [Gao86, Gao90, SG91]. Most of these subclasses are reviewed throughout this document. In general, the strategy behind Sarkar and Gao's techniques is to use graph coloring to analyze the suitability of a collection of loops to undergo collective transformation. When a cluster of loops can be found which is compatible, the individual loops can be fused, allowing several follow-on transformations to be performed.

The loop transformations employed in identifying and analyzing loop clusters are loop normalization, direction reversal, and loop interchange. *Loop normalization* transforms the loop-control portion of each loop so that all loops in the cluster have uniform bounds, step, and direction. *Loop reversal,* which is often necessary for loop normalization, reverses the order of array element access within a loop body by reversing the indexing function of array references. This transformation is necessary whenever the direction of a loop is changed. *Loop interchange*, on the other hand, changes the order of array referencing by changing the order of nesting of perfectly nested loops In later chapters, illustrations are given of each of these transformations.

Once a compatible loop cluster is identified, loop transformation is performed to improve program performance Among the follow-on loop transformations which are used to accomplish this objective are loop fusion, array contraction, and software pipelining. *Loop fusion* is accomplished by merging statements from several loops, each having the same loop direction, into a single loop body. The effect is to reduce redundant loop control while increasing loop-body size. With a larger loop body, the need for loop unrolling and the effectiveness of instruction scheduling are also effected Without fusion, loop unrolling is often used to increase the number of independent instructions within a loop body, available for instruction scheduling *Loop unrolling* places statements from several loop iterations within a single iteration, increasing the loop step commensurately. *Array contraction* transforms array element references to equivalent scalar references. This transformation can only occur if an array element is both produced and consumed within the same loop body [1] Contraction has the effect of eliminating array indexing instructions as well as the associated long-latency loads and stores that accompany such references. At the same time, elimination of these array references reduces storage requirements by eliminating the arrays themselves. The impact of array contraction can be substantial, as we will show. *Software pipelining* is an instruction scheduling technique which through an unrolling process finds a pattern of independent instructions across iterations, all of which can be simultaneously executed if the host machine has sufficient parallelism. Not only does this transformation increase the number of instructions which are concurrently executable, it distances long-latency dependencies which otherwise might stall the hardware execution pipeline. As with the preceding transformations, these transformations too are illustrated in subsequent chapters.

In cases in which loop compatibility does not exist, the constituent loops of a cluster can usually be partitioned into smaller compatible clusters, and follow-on transformations, such as loop fusion, applied to each smaller cluster In doing so, much of the original benefit from loop fusion can still be achieved. Loop partitioning

[1] Wolfe uses the term "array contraction" to refer to the technique for reducing the size of a compiler generated temporary array created as a result of scalar expansion [Wol89]

for loop fusion is the subject of recent research by Gao, Olsen, Sarkar, and Thekkath, and many of the ideas presented in the document derive from their research, although the specific algorithms we employ are slightly different in several instances [GOST92]

1.2 Thesis

The proposition of this thesis is that it is both feasible and beneficial for an optimizing compiler to perform collective loop analysis. To prove this proposition, we first show how collective analysis can be efficiently performed on compatible loop clusters. Our primary focus in this regard is the application of loop analysis to programs written in traditional imperative programming languages, such as FORTRAN, in a uniprocessor setting in which the compiler optimization strategy is to fuse loops within each compatible loop cluster. Although the loop analysis technique we describe applies only to loops with certain restricted classes of array references, these classes include the most common cases that occur in practice, for example, the cases of single dimension and two-dimensional arrays. We also describe a minimum cost partitioning heuristic to achieve efficient fusion in the not so unlikely event the compiler encounters a non compatible loop cluster. Although a difficult problem to solve, many common cases can be solved efficiently with optimal results, as we will show

Through a series of timing tests on various types of actual uniprocessors, we also show the extent of performance improvement possible from a transformed cluster of compatible loops. We assess the effect of efficient and naive partitioning on a non-compatible loop cluster. For these tests we use a scalar RISC workstation (A SPARC Server 4/490), a superscalar workstation (An IBM RISC System 6000, Model 550), and a mainframe computer with an attached vector processing facility (An IBM 3090/VF). For each type of machine, loop transformation is shown to be of significant benefit. Moreover, this benefit is shown to be equally important in the case of non-compatible clusters To further substantiate our results, we replicated our experiments on a scalar processor simulator and a cache simulator These tests reveal yet additional insights into the effects of the individual transformations, while allowing us to isolate instruction-scheduling effects Overall our tests confirm that collective loop analysis and transformation can be of substantial benefit in optimizing compilers for the three uniprocessor architectures that were used Moreover, we produce evidence to suggest that the performance benefit from these transformations, especially array contraction, can be substantial, resulting in a four-fold speedup in processor performance in certain instances.

1.3 Overview

There are four main parts to this document: The first part describes the process of loop-cluster analysis, first in the case of compatible clusters and then in the case of non-compatible clusters. The second part describes related work on this subject, along with a list of topics for further research. The third part of the document describes essential transformations needed to perform collective loop analysis, along with several common follow-on transformations intended to improve processor performance. This part also describes a number of important related transformations which are used in the fourth part of the document as a basis for performance-test comparison. And, the fourth and final part then describes the results of timing and simulation experiments conducted to evaluate the performance of collective loop transformation. As an addendum to this document, we also include several appendices, the first contains sample code listings used during loop-transformation performance evaluation, and the remaining appendices each provide a brief description of one of the computer architectures and compilers used during the evaluation.

Our discussion of loop analysis begins in Chapter 2. In this chapter we first outline the preconditions under which collective loop analysis can be performed. We then describe a variant of Sarkar and Gao's graph-coloring technique which can be used to determine whether the constituent loops within a loop cluster are compatible for collective transformation. A description of the two principle graphical constructs used during this compatibility analysis is given, along with the basic intuition behind the technique. The two principle constructs we describe are *1)* the *Loop Communication Graph* (LCG) and *2)* the loop compatibility *Interference Graph* (IG). Justification for the loop analysis methodology is then provided along with a detailed description. An example of the analysis of a cluster of non-nested loops is then provided, along with a motivating description of several basic follow-on transformations that might be performed to increase loop-cluster performance. The chapter then continues with an example of another special case, one involving doubly-nested loops, and finally it concludes with a brief description of the strategy for handling clusters of loops which have an arbitrary, but uniform, level of nesting.

In Chapter 3 we examine the not unlikely case in which compatibility does not exist among the loops of a cluster. In this case, compatible subsets of loops must be identified through a process of cluster partitioning. To motivate our discussion, we first investigate the nature of non-compatibility. Based upon our analysis of some simple situations in which non-compatibility exists, we identify factors which ultimately determine the optimality of cluster partitioning. We then apply our observations to several brief examples which allow us to gain yet a better understanding of the requisite conditions for optimal partitioning. Based upon these examples we derive an efficient heuristic to partition a non-compatible cluster of loops into smaller compatible clusters, using the *flow-augmenting path* algorithm from network flow theory. We then conclude this chapter with a complete description of our methodology and the rational behind it.

Chapter 4 describes related research in the area of collective loop transformations. This work is especially important because much of the independent research reported in the previous two chapters derives in one way or another from the research reported here. We begin the chapter with a description of Sarkar and Gao's methodology for collective analysis, describing both its advantages and disadvantages. One important aspect of this particular research is its applicability to multiprocessor situations, a situation we ourselves do not directly address. Next we describe an alternative algorithm for cluster partitioning, called *collective loop fusion*. As part of our description we compare collective loop fusion with the partitioning technique described in Chapter 3. We then follow this description with a brief survey of related research in the area of loop transformation, and finally we list a number of suggested topics for further research.

Chapter 5 contains a discussion of related loop transformations which are essential *1)* during the screening process to identify eligible loops, *2)* during the analysis process to achieve loop normalization prior to analysis and pipeline compatibility following analysis, and *3)* during performance testing, to provide a basis for determining loop transformation effectiveness. Among the transformations discussed in this first category are in-line expansion, normalization, scalar renaming, and node splitting. Transformations which fall into the second category are direction reversal, loop interchange, loop fusion, array contraction, and software pipelining. And, transformations which fall into the third category are instruction scheduling and loop unrolling.

The focus of Chapter 6 is on performance testing. The chapter itself is divided into two parts, the first part discusses timing tests, and the second part, simulation results. For each part we evaluate transformation performance based first upon a compatible loop cluster and then upon a non-compatible cluster. In the first instance we are primarily interested in performance improvement from the respective loop transformations themselves, while in the second instance we are interested in the impact of alternative partitioning choices In the second part of the chapter, we use processor and cache simulations to evaluate several factors which could not be easily isolated by timing tests alone, specifically, the impact of instruction scheduling and of cache We conclude our chapter on performance tests by summarizing key findings, based upon our overall test results.

The final chapter, Chapter 7, is an attempt to tie everything together. In the first part of the chapter we summarize the most significant findings from our research, and the the later part we list additional topics we feel might lead to important new results in this area

As noted previously, we conclude this document with a number of appendices containing material specifically related to the performance testing described in Chapter 6. Appendix A contains sample listings of the code used for the actual timings, and Appendices B to D describe the various architectures and compilers used during both the timing tests and the processor/cache simulations.

Chapter 2

Analysis of Loop Clusters

Much of the computation involved in scientific programs occurs within loops which produce or consume large arrays. As a result, being able to handle loops efficiently is of fundamental importance. Much work has been accomplished within the past several years with regard to loop optimization, and many effective techniques have emerged [Ban88, KRP+81, PW86, Wol89, WL91]. These techniques usually focus on reordering statements either to avoid data and control dependences which can stall a hardware execution pipeline or to avoid long latency memory operations. A key point, however, is that since these techniques usually pertain to a single loop nest, their effect is necessarily local.

In this chapter a global analysis technique is described to find a *compatible* set of loops across a cluster of loops. By compatible we mean there exists a uniform sequence of array references which allows a cluster of loops to be processed in a software pipelined manner. A compatible order might be found from ascending and/or descending loop directions or, in certain circumstances, from the interchange of two levels of loop nesting. We refer the the process of finding a compatible set of loops as *collective loop analysis*. In general, collective analysis makes use of several common loop transformations, among which are loop normalization, direction reversal, and loop interchange. If loop analysis is successful, in other words, if a compatible set of loops can be found, several other common transformations to improve the loop cluster become possible, as we will show with a series of brief examples, the primary transformations being loop fusion and array contraction. Other code improving transformations, such as software pipelining, loop unrolling, and instruction scheduling, might also be used to further enhance the quality of the overall code, but in order to focus on the analysis phase of loop processing, we leave description of these latter transformations, along with complete descriptions of the former, until Chapter 5.

Overall, this particular chapter is organized as follows. In the first section we describe relevant conditions, or preprocessing, necessary for collective loop analysis. We also describe the graphic constructs used during compatibility analysis, along with related terminology We then explain the general rational behind compatibility analysis and describe the technique we use to accomplish this analysis, a technique which is similar, but not identical, to the one proposed by Sarkar and Gao [SG91] In the section that follows, we give an example of the collective analysis of the common case of non nested loops We then cover analysis of doubly-nested loops, and finally, in the last section, we examine extensions of collective loop analysis that are appropriate for other special cases involving higher-level nested loops

2.1 Preprocessing

Several conditions are necessary to qualify loops for analysis, and selection of loops satisfying these conditions forms an integral part of the collective analysis process. In selecting loops, the following restrictions apply:

1. The dimensionality of the arrays must be the same for all arrays referenced within the cluster of loops being analyzed.

2. The index variable of all loops must increase or decrease by the same constant amount Usually this amount, called the loop increment, or *step*, is ±1.

3. The loops used to define the arrays must be perfectly nested, and the number of loops within each nest must equal the dimensionality of the arrays being referenced For example, a code block which performs matrix multiplication on conformable square matrices is disallowed because the calculation of the inner product requires an additional loop, one more than the dimensionality of the participating arrays

4. An index variable can only appear in one subscript position of an array reference. Therefore, references of the type `A(I,I+J)` (in which I and J are the respective induction variables for a nest of two loops) are disallowed This condition, that subscripts be *uncoupled*, also ensures that each reference has the same number of subscripts as there are loops in the surrounding loop nest.

5. Although not essential, another assumption we make is that each array element be defined (that is, stored) only once. This condition, called the *single-assignment rule*, is a common characteristic of functional and applicative languages, such as Haskell and SISAL, and it is implicit in array statements used in FORTRAN 90.

The above criteria restrict the possible type of loops which might be included in a cluster to only those which can be feasibly transformed. These restrictions and other program characteristics make code conditioning essential. The following paragraphs describe some of the conditioning that should occur. First, function in-lining should be performed on any function that makes reference to a globally defined array. The purpose of in-lining is to collect together as many potential loops as possible. This is important because structured programming results in code which uses lots of functions. In-lining should be followed by data flow analysis. Intraprocedural flow analysis ensures that scalar variables that are referenced within the loop bodies are not redefined between their use in one loop and their reuse elsewhere in any follow on loop. What intraprocedural flow analysis implies, in this case, is that any code found between eligible loops can be moved out of the way without affecting program correctness. The next step in collective analysis should be loop normalization. The purpose of normalization is to provide a uniform basis for loop comparison. Normalization consists of *1)* setting the loop control within each loop to a range which extends from a lower bound of 1 to an upper bound of N, where N is the adjusted bound, and *2)* setting the step to 1. Of course, any changes made to loop parameters also necessitate corresponding changes to index variables used within the respective loop bodies.[1]

Once normalization is completed, loop selection can be performed. Loop selection involves *1)* array verification, *2)* loop bounds checking, and *3)* nesting level checking. Array verification ensures that only those loops which actually produce or consume a globally declared array are included in the cluster. While verification is occurring, subscript coupling could also be checked (Item 3 above). Bounds checking, on the other hand, ensures that the loop upper bound for each loop control statement is the same for all loops under consideration. Lastly, a nesting level check ensures that all loops under consideration have the same level of nesting. As a final phase of preprocessing, local flow analysis should be performed to identify data dependence within loop bodies, dependencies which constrain loop direction, thus preventing direction reversal.[2] Loops which posses data dependencies would then be tagged

[1] Loop normalization is described in Chapter 5, Section 5.2.1 (see page 78)

[2] Direction reversal is described in Chapter 5. Section 5.4 (see page 82)

as "constrained". Once data flow analysis is completed, the resulting loop cluster selected would be ready for loop compatibility analysis. Compatibility analysis, which is the primary topic of this chapter, is described next.

2.2 Graphical Constructs

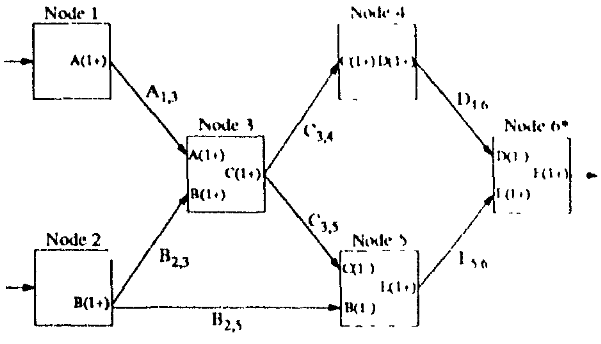
To understand the compatibility analysis process, one first must understand the tools, or constructs, used during the process, the two most important constructs employ graphs. Graphs are often used to depict relationships among individual entities in a way which makes a global relationship clear. A case in point is graph coloring. The classic example is the coloring of, say, political regions on a territorial map. Coloring problems such as map coloring are solved using an *interference graph*. In the case of a map, the regions become nodes and adjacent regional boundaries become arcs. The objective in this case is to color the map interference graph so that no two adjacent nodes are the same color. In general, determining a minimum, or *chromatic number*, is difficult (an NP-complete problem [PS82]). However, in many situations, coloring heuristics have proven quite practical, as in the case of register allocation [CAC+81].

What Sarkar and Gao discovered is that the analysis of loop clusters can also be converted to a graph coloring problem since an analogous correspondence often exists with respect to the orders in which array elements are defined, or generated, within some loops and the order in which they are used, or consumed, in later loops. The objective of coloring in this particular instance is to find a set of compatible loops which produce and consume array elements in a compatible order which allows array values to flow through a cluster of loops in a manner similar to software pipelining.

Collective loop analysis begins with the construction of a *Loop Communication Graph* (LCG). An LCG is a directed multigraph depicting the flow of array values between the code blocks that produce and consume arrays. The "flow" in this case represents flow dependence between assignments to array elements in one code block and the corresponding uses of these elements in follow-on blocks within the loop cluster. Formally, an $LCG = [N, A \quad A \subseteq N \times N]$, where $n \in N$ is a node and $1 \leq n \leq |N|$. A node n in an LCG represents a code block comprised of perfectly nested loops [Wol89]. An arc $X_{m,n} \in A$ represents the transfer of array elements from a producing code block to a consuming one. Graphically, this transfer is shown by connecting the origin of arc $X_{m,n}$ to the output port of m corresponding to array X and the terminus of $X_{m,n}$ to the respective input port of n. An example cluster of

```
DO I = 1, N
   A(I) = I
END DO
DO I = 1, N
   B(I) = N - I
END DO
C(1:N:1) = A(1:N:1) + B(1:N:1)
D(1:N:1) = C(1:N:1) + 99
E(1:N:1) = C(N:1:-1) + B(N:1:-1)
DO I = 1, N
   F(I+1) = F(I) + D(N-I+1) + E(I)
END DO
```

(a) A Collection of FORTRAN-90 Loops



(b) Loop Communication Graph (LCG)

Figure 2.1: Graphic Representation of a Collection of Loops

loops and corresponding loop communication graph is shown in Figure 2.1 Nodes in an LCG are further specified by a tuple of Array Access Vectors

Each *Array Access Vector* (AV) is itself a vector of tuples in which the first element of the tuple indicates the correspondence between the subscripts used to access an array and the loops used to generate these subscripts; the second element indicates the direction of the corresponding loop. A direction element can be either positive (increasing) or negative (decreasing). A positive element is indicated by a plus (+), and a negative element, by a minus (−). An AV contains one loop-direction tuple for each subscript in an array. For example, A(2+,1−,3+) describes an array A which is referenced within a three-loop nest in which the outer loop causes the second subscript to decrease, the middle loop causes the first subscript to increase, and the inner loop causes the third subscript to increase. An AV $n_x = \{(i, \{'+', '-'\})^k \mid 1 \leq i \leq k\}$ is associated with each input and output port x incident to node n in the LCG, where i is is a loop identifier and k is the depth of loop nesting When the AV m_x for an array X which is produced in code block m is paired with the AV n_x for an array of the same name which is consumed in another code block n the pair form the specification of the arc $X_{m,n}$ in the LCG. (See Figure 2.1(b))

For certain code blocks the loop direction cannot be reversed because of a data dependence across iterations of the loop. Such a dependence is usually referred to as a *loop-carried dependence*. In this case the corresponding node is called a ***constrained node***. The node for the loop-carried dependence created by the assignment statement in the last code block in Figure 2.1 is an example. In an LCG, nodes which are marked with an asterisk refer to constrained nodes, as indicated for Node 6 in Figure (b).

Collective analysis applies to many different loop configurations, each representing a different array-reference sequence; but the methodology we describe considers only two mutually exclusive array-reference sequences at a time. For loops involving vectors (i.e., one-dimensional arrays), the sequence is either ascending or descending loop directions, and for doubly-nested loops that are uniformly ascending, or descending, alternative sequences are determined by interchanging two nested loops within the block, or equivalently, switching the sequence of indices within all access-order vectors of the corresponding block. The way this works is described fully later in the chapter. For now, however, we merely emphasize that only two array-reference orders, or sequences, are allowed at a time, for each block during each phase of loop compatibility analysis, either *1)* alternative loop directions or *2)* an interchange of two levels of loop nesting. The information about these alternatives is contained in the *Interference Graph* of the loop cluster used during the particular phase of analysis.

An Interference Graph (IG) is an undirected multigraph having nodes grouped according to *blocks*, each block being comprised of two nodes: a ***primary node***, n^+, corresponding to a node in the LCG, with input and output ports matching the respective origins and terminuses of incident arcs to this node, and a ***compliment node***, n^-, containing a copy of the primary node but with AV elements changed to reflect the alternative array-reference sequence allowed. When loop direction is used as the basis for analysis, the sign of the elements in the AVs are reversed; otherwise, when nesting level is used, corresponding elements in the the AV tuple are switched. For example, the first block, corresponding to Node 1 in the LCG, shown in Figure 2.1(a), consists of two nodes; a primary node n_1^+, labeled "Node 1" with AV `A(1+)`, and directly below it, a compliment node n_1^-, labeled "Node −1" with AV `A(1-)`. Formally, an $IG = [[N,\overline{N}], E : E \in N \times N]$, where $[N,\overline{N}]$ is a block, $n^+ \in N$ and $n^- \in \overline{N}$ are nodes within a block, for $1 \leq n \leq |N|$. An undirected edge $X_{m,n} = (m^+, n^\pm) \in E$, called an ***interference edge***, represents non-compatibility between the order of generation (definition) of array values in one code block and the order of their consumption (use) in another.

Obviously a code block cannot simultaneously process array elements in more than one order at a time. Therefore, to reflect the fact that alternative reference

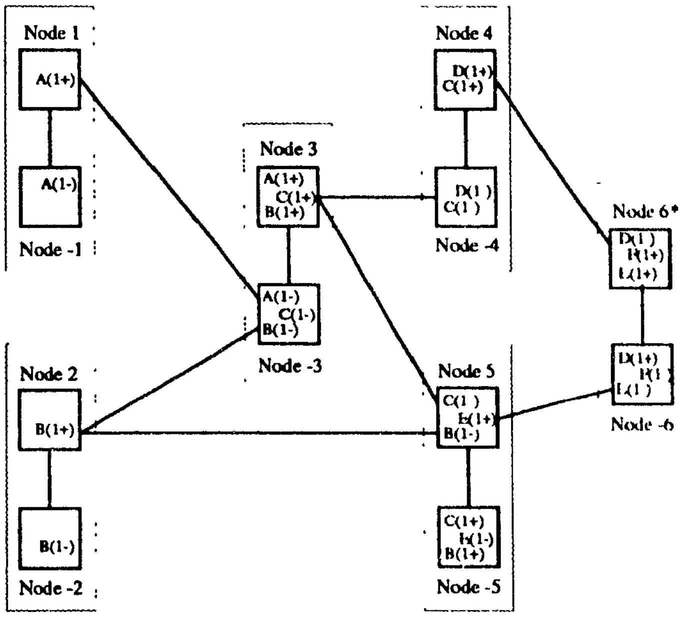


Figure 2.2: An Interference Graph (IG)

orders represented by the primary and compliment nodes within a block are not simultaneously possible, we attach an interference edge between the two. Likewise, we attach an edge between the output port of a node in one block and input port of a node in another when the two ports reference the same array and the corresponding AVs in the two nodes differ. In this second situation, the edge indicates a situation we would prefer to avoid, since unless the AVs are the same, the array elements which pass between the two nodes, or code blocks, cannot be uniformly passed from one block to the other, as we would like.

Note that not all interference edges need to be attached, since the edges extending from the primary nodes to the nodes in follow-on code blocks provide sufficient interblock compatibility information, by themselves, to allow us to do our analysis. Attaching the corresponding arcs from the complement nodes to follow on blocks therefore becomes unnecessary. For example, notice that, in Figure 2.2, an edge has been attached between Nodes 1 and −3 to indicate non-compatibility in the array-reference orders for AVs A(1+) in Node 1 and A(1-) in Node −3, while no arc was attached between Nodes −1 and 3, even though the non-compatibility of the complement AVs, A(-) and A(+), respectively, is equally valid. The absence of "redundant" edges from the complement nodes in each block not only reduces the number of linkages which must be made to the graph, but it also simplifies the ensuing analysis, as will become evident

Algorithm I *Construct an Interference Graph (IG).*

input: An LCG = $[N, A]$ in the form of a linked list; each node in the list corresponds to a code block. The nodes n are each specified by a set of input and output ports, n_x Each port refers to an AV for some array X referenced within the block, along with list of connecting nodes m indicating from which node array X was produced or to which m it is being sent.

output: An IG = $[[N, \overline{N}], E]$ created from the LCG; each block in the LCG consists of two nodes: a primary node n^+ containing the same input and output ports as corresponding node n in the LCG and a complement node n^- containing the same input and output ports, but with AV's n_x, $\forall x \in n$, switched to indicate the alternative array-reference order (see Figure 2.2).

procedure:

```
Create a complement field in each node.
    FORALL $n \in N$ in the LCG
        copy $n$ to primary node $n^+$ and to complement node $n^-$
        FORALL $n_x \in n^-$
            switch the direction of AV elements
        insert nodes $n^+$ and $n^-$ into the graph
Add interference edges.
    FORALL $n^+ \in N$
        FORALL output ports $n_x$
            IF $m_x$ for node $m$ specifies the alternate vector
                attach a link to $m^\pm$ creating $X_{n^+m^\pm}$
            ELSE
                attach a link to $m^\mp$, creating $X_{n^+m^\mp}$
```

2.3 Justification and Methodology

Were we to consider all possible array-reference orders possible from a cluster of loops, there would be $(k! \cdot 2^k)^n$ such possibilities, where n is the total number of AVs used within the code blocks of the loop cluster and k is the level of nesting of these loops. Obviously, analysis of so many alternatives at a time would be intractable for all but the simplest loop clusters. However, by restricting analysis to just two orders at a time, say, ascending and/or descending loop direction, problem size is significantly reduced. Moreover, by considering only two alternative orders at a time, we can easily represent any of the 2^n remaining possibilities with a just single IG (again assuming n is the number of AVs). From construction of the IG, we know that neighbor nodes always reflect a complementary array-reference order, and for an overall compatible order to exist, it must consist of orders taken from alternating nodes. Consequently, the IG must be a bipartite graph if a compatible set of loop directions is to exist. Although the number of possibilties can still be very large, we know that there can now be at most two compatible orders for any cluster overall. This fact is what finally makes the problem tractable, since to determine whether a compatible loop ordering exists, we need only determine whether the particular IG under analysis is, in fact, a bipartite graph.

To take advantage of this bipartite property, we look to a few simple notions from graph theory [BM76, Deo74, Har69], and we begin under the premise, based upon our above observations, that *an IG is bipartite if, and only if, an overall compatible set of array-reference orders exists.* Accepting this fact, it is now easy to show that any bipartite graph is two-colorable (the proof by construction is trivial): Color elements of one partition one color and elements of the remaining partition a second color. Likewise, it follows immediately from the general definition of a spanning tree that the spanning tree induced by an IG represents a minimal connection of nodes within that IG. Therefore our next step to show that every nontrivial (spanning) tree (induced from an IG) is two-colorable.

Theorem 2.1 *Every nontrivial tree is two-colorable.*

proof: (by induction)

basis: Assume T is a tree with two nodes. If one of the nodes is colored with one color, the only remaining node can be colored with a second color *inductive step.* Assume T contains n nodes that have already been successfully colored with two colors If another node is added, regardless of its location, it can always be colored an opposite color to the color of the node to which it is attached. □

Once a spanning tree is formed within the IG, we are assured of creating a simple cycle for each of the remaining original arcs added to tree Fortunately determining whether the cycles formed in this way are themselves two-colorable is quite easy; we merely inspect the origin and terminus of each edge being added, and if the two have different colors, edge insertion can be safely accomplished, preserving the colorability of the graph If edge insertion is successful for all remaining edges, we know the IG overall is two-colorable, thus confirming the fact that the IG is a bipartite graph and that a set of compatible loops exists Now to find this order we need only select a set of nodes of the same color. This is because the way we constructed the graph assures us of a direct correspondence between compatible loop configurations and node coloring, that is, we know for a compatible cluster of loops, opposite nodes of every interference edge are both an opposite loop configuration and an opposite color.

The preceding analysis suggests how the determination of loop compatibility can be accomplished. The approach we describe is based upon creating a two-colored spanning tree during a breadth-first walk of the IG, commencing at the output block of a loop cluster. Our particular approach (which is slightly different from the one proposed by Sarkar and Gao) has a special property in that it identifies the least costly non-compatible interference edges to remove whenever no overall set of compatible array-reference orders exits, and it does so efficiently, as we will show in Chapter 3. The cost to which we refer is the cost of referencing elements of an array from memory, and the least-cost edges are those corresponding to arrays that are used by multiple blocks, not just one. The non-compatible edges, if any are found, ultimately form part of a cut-set used to fird compatible clusters of the original cluster which can ultimately be fused, allowing elimination of all arrays except the ones that cross a cluster boundary. We discuss *cluster partitioning,* in detail in the next chapter.

Again, the basic strategy behind compatibility analysis is to create a two-colored spanning tree during traversal of the IG. We proceed by first identifying and coloring the primary node of the output block of the IG. Then, starting with this node, we traverse the remaining nodes in the gr ph, in a breadth-first manner. At each node visited during our traversal we examine both the input ports and the port to the complement node within the same block, in each case inspecting the terminus of the interference edge. If the terminus is uncolored, we color it the opposite color of the current node If, on the other hand, it has already been colored, but the color is the same as the color of the current node, we note the fact that the cluster is non compatible and store a reference to the non-compatible interference edge to be used during subsequent cluster-partitioning analysis. After all nodes have been reached, the algorithm stops. Nodes that are of the same color represent a compatible set of loops. Of course, this leaves a choice of the two colorings. One guideline regarding which color nodes to choose is to select nodes that correspond to the color of most of the "primary" nodes, since doing so involves the least follow on direction reversal (the direction-reversal transformation itself will be explained shortly). On the other hand, choosing a color might not be an option at all if the graph contains constrained nodes, since the color of any constrained nodes will uniquely determine the color set which must be used, that is, of course, assuming all constrained nodes are of the same color. Otherwise, if the constrained nodes are not all of the same color, the cluster will not be compatible overall. Our procedure for testing loop compatibility is summarized in the following recursive algorithm:

Algorithm II *Determine a set of compatible array-reference orders for a cluster of loops, or verify that a compatible set of loops does not exist.*

input: An IG $= [[N, \overline{N}], E]$ in the form of a linked list; each node in the list is either a primary node n^+ or a compliment node n^- (see Figure 2.2),

A **colorset** = {color$_1$,color$_2$}, and a **color_preference** which determines the color of the set to be chosen in the event the loop cluster contains constrained nodes; initially color_preference is undefined,

A **checkset** which initially contains the primary node of the output block of the cluster; we refer to this node as **root**, and **root color** = color$_1$,

A **nextset** = $\emptyset$, representing the next level of nodes to visit,

A set of **noncompatibles** = $\emptyset$, which contains any edges identified as having a non-compatible array-reference order with that of its neighbor node (later, we refer to these edges as X-edges), and

A Boolean return value, compatible, with an initial value of true

output: An IG with compatible nodes indicated by the primary color in the color set; however, if any nodes are constrained, the compatible nodes will be those with the same color as the color_preference.

procedure:

```
FUNCTION colorTree (checkset):   compatible
FORALL $n \in$ checkset
    FORALL input arcs $X_{mn}$ and the arc to complement node $\overline{n}$
        IF the terminus $m$ is not colored
            color $m$ the opposite color of $n$
            add $m$ to nextset
        ELSE IF $m$ is the same color as $n$
            add $X_{mn}$ to noncompatibles
            compatible = false
        IF $m$ is constrained
            IF color_preference has not been set
                set color_preference to the color of $m$
            ELSE IF the color of $m \neq$ color_preference
                add $X_{mn}$ to noncompatibles
                compatible = false
        IF nextset $\neq \emptyset$
            compatible = colorTree(nextset)
    RETURN compatible
```

The graph in Figure 2.3 shows the IG in Figure 2.2 after it has been colored using Algorithm II. The dark edges define the spanning tree created during algorithm execution, and the shaded nodes indicate the set of compatible loops. Note that the constraint on Node 6 prevents the unshaded nodes from also forming a alternative set of compatible directions for the respective loops.

Algorithm II handles situations in which there is only a single output block in the LCG, but situations in which there are multiple output blocks often occur which require a slightly more complex solution, one involving the use of multiple color sets—or rather a different color set for each of the output blocks within the original cluster. In this case, a primary node from each output block is selected to begin the algorithm, and each root node is colored, as in Algorithm II—each with a color from its own respective color set Then, as before, a depth-first walk is performed beginning from each root. If the LCG is connected (and we can assume that it is), there will come a point where a node is reached that has already been colored, but with a color from a

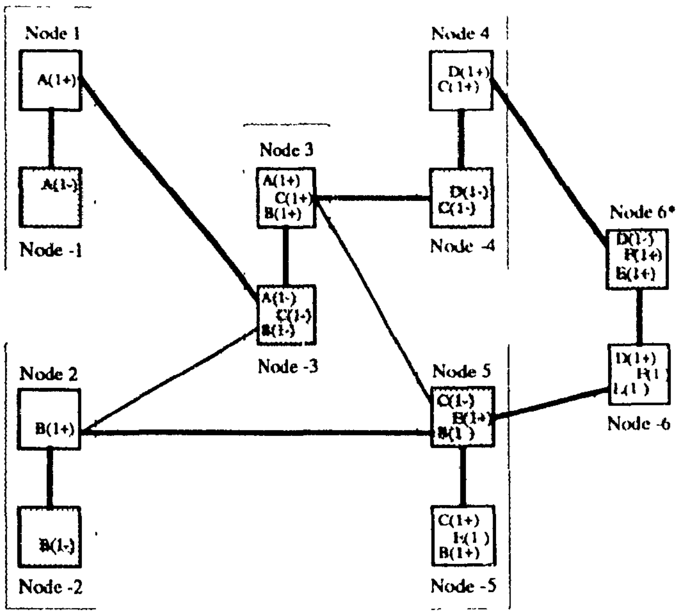
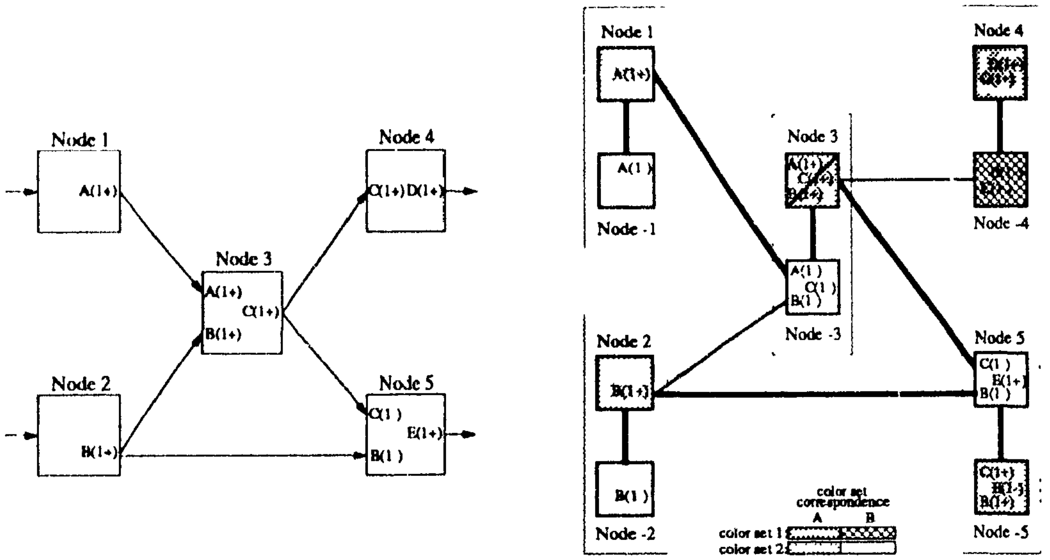


Figure 2.3: An Interference Graph after Coloring

different color set. When this occurs, the corresponding colors from each of the two sets are henceforth treated as though they were one. Then, should further traversal lead to some later convergence of the two color sets, at the point of terminus checking from some node, the decision whether to add the new edge, or not, will be based upon the initial color-set equivalence established when the color sets first converged. An illustration of the coloring of a loop cluster with two output code blocks is shown in Figure 2.4; the LCG is shown in Figure 2.4(a) and the corresponding IG after coloring is shown in Figure 2.4(b).

2.4 Follow-On Transformations

In the last section we described in general terms the collective analysis of the example cluster of loops given in Figure 2.1 (see page 12). To put this analysis into perspective, we now step through the basic actions taken once a compatible set of loop directions have been found. We begin our follow-on description with the coloring shown in Figure 2.2 (page 14). Notice that Node 6 is constrained because of a loop carried dependence of array F. As a result, there is only one set of compatible sequences from



(a) Loop Communication Graph **(b) Colored Interference Graph**

Figure 2.4: Loop Collection having Multiple Output Blocks

which to choose, instead of two as would have been the case had the loop-carried dependence of F not existed. This set is the one indicated by the shaded nodes in the figure— the cluster consisting of Nodes −1, −2, −3, −4, 5, and 6.

Once the set of compatible nodes have been selected, we apply direction reversal to the loops corresponding to the negative nodes in the set, i.e., the nodes having a negative node number. The basic notion behind direction reversal is merely to substitute N-I+1 for each occurrence of I in the loop body. In this case, applying direction reversal to the first four loops of the normalized code, shown in Figure 2.1, results the transformed code shown at the top of the next page, on the left-hand side. Since normalization ensures uniform loop control and since coloring analysis ensures loop compatibility, we know too that the loops in our example cluster can now be safely fused, as shown at the top of the next page, on the right-hand side.[3]

[3] Direction reversal is described in Chapter 5, Section 5.4 (page 82), and loop fusion is described in Section 5.7 (page 84).

After Direction Reversal

```
DO I = 1, N
   A(N-I+1) = N - I + 1
END DO
DO I = 1, N
   B(N-I+1) = N - (N-I+1)
END DO
C(1:N:-1) = A(1:N:-1) + B(1:N:-1)
D(1:N:-1) = C(1:N:-1) + 99
E(1:N:1) = C(N:1:-1) + B(N:1:-1)
DO I = 1, N
   F(I+1) = F(I) + D(N-I+1) + E(I)
END DO
```

After Loop Fusion

```
DO I = 1, N
   A(N-I+1) = N - I + 1
   B(N-I+1) = I - 1
   C(N-I+1) = A(N-I+1) + B(N-I+1)
   D(N-I+1) = C(N-I+1) + 99
   E(I) = C(N-I+1) + B(N-I+1)
   F(I+1) = F(I) + D(N-I+1) + E(I)
END DO
```

The final transformation, for now anyway, is array contraction, and the following code is the result of applying contraction to the previously transformed code:[4]

Array Contraction

```
DO I = 1, N
   a = N - I + 1
   b = I - 1
   c = a + b
   d = c + 99
   e = c + b
   F(I+1) = F(I) + d + e
END DO
```

Notice that array contraction replaced, by a scalar variable, all array references in which an element reference was first produced then later used. For example, `A(N-I+1)` is replaced by **a**, and so forth. Since all but the last array was produced then consumed in an element-by-element manner, the need for intermediate arrays, as in the original code, no longer exists. In later chapters we will show that yet additional transformation is not only possible, but, beneficial

The effect of collective transformations, such as these, can be dramatic, loop overhead and memory requirements can be reduced, and most array indexing operations can be eliminated. Moreover, a single larger code block results which facilitates instruction scheduling. In Chapter 5 we describe yet additional transformations which

[4] Array contraction is described in Chapter 5, Section 5.8, (page 5.8)

might be performed or which can be affected by collective transformation, and in Chapter 6 we provide experimental evidence to show the extent of performance improvement attributable which is actually realizable on each of several different types of uniprocessor architecture

2.5 Analysis of Doubly-Nested Loops

So far we have only discussed the situation involving single-nested loops on one-dimensional arrays since this situation is clearly the most intuitive and easiest to handle. The difficulty with multi-nested loops and multi-dimension arrays stems, of course, from the enormous increase in complexity which arises from the many permutations of the levels of loop nesting and loop direction Whereas in the non-nested case AVs have only two possible orders, in doubly-nested cases there are $2^2 \cdot 2!=8$ possible sequences. In spite of this enormous increase in complexity, the basic methodology described in the previous section still applies—that is, with a few appropriate enhancements. Recall from the previous section, our strategy was to identify an alternative, or complementary, set of reference orders for each AV in the loop cluster, and then analyze each of the enumerated possibilities using the graph coloring technique described in Algorithm II. In this section we show how this same methodology can be applied to certain situations involving clusters of doubly-nested loops and two-dimensional arrays.

In the case of doubly-nested loops, the compiler faces essentially three possible possibilities: *1)* the direction of all loops are the same across the entire cluster of loops, but the order of the array subscripts in each reference varies, *2)* the order of the subscripts are the same across the entire cluster of loops, but the direction of the corresponding loops in the cluster varies, and *3)* a combination of these two situations in which both the order of the subscripts and the direction of the loops vary across the cluster of loops.

CASE 1 *The direction of the loops are the same across the cluster of loops, but the order of the subscripts varies.*

In this situation, AVs for the arrays referenced within the cluster are either of the form *array_name(1+,2+)* or *array_name(2+,1+)* or they are of the form *array_name(1−,2−)* or *array_name(2−,1−)*. In either situation, the direction of the loops are presumed to be compatible (so no direction transformation is required).

On the other hand, the order of the array subscripts creates the bipartite condition required for two-coloring the IG. In this instance we use Algorithm II, as before, but create the complementary nodes using AVs with a subscript order opposite to the subscript order of the AVs in the primary nodes. Also, instead of using loop reversal, as we did previously, we use loop interchange to align array access orders consistent with the results of the coloring analysis (Loop interchange is described more fully in Chapter 5, Section 5.5)

CASE 2 *The order of the subscripts are the same across the cluster of loops, but the direction of the loops varies*

The AVs in this case are either of the form *array_name(1±,2±)* or the form *array_name(2±,1±)*. To analyze these loops, we treat the problem as though it were two separate problems, analyzing the same AV element from each AV as though the referenced arrays were one-dimensional. For example, we might first analyze the directions of the outer loops of the code blocks in the cluster, and if a compatible direction is found, use the results of this analysis to analyze the directions of the inner loops (or visa versa). As in our original examples, the transformation technique used to bring the unaligned loops into alignment is loop reversal

CASE 3 *Both the order of the subscripts and the direction of the loops vary.*

This last situation is a combination of Case 1 and Case 2. To analyze loop clusters of this type, we first attempt to color the transformed LCG based upon subscript order, as described in Case 1, and if the coloring is successful, we continue by attempting to color the LCG based upon loop directions, as described in Case 2

An example of the combined approach is shown in Figures 2.5 2.8 The cluster of doubly-nested loops used in this example are based upon the code segment shown in Figure 2.5 The corresponding LCG for this code is shown in Figure 2.6(a) The loop analyzer of the compiler uses this LCG during the first phase of its analysis to create the IG shown in Figure 2.6(b). Phase-1 analysis is merely the Case 1 analysis, described above. Notice that since the compiler's first attempt at transformation involves loop interchange, the complementary nodes in the Phase 1 IG are composed by copying the primary nodes, which were taken from the LCG, and reversing the of order of their AV elements (as opposed to changing the direction of these elements as was done in the case of non-nested loop clusters). The result of this analysis, a two coloring of the IG, is also reflected in Figure 2.6(b). Although the coloring produces

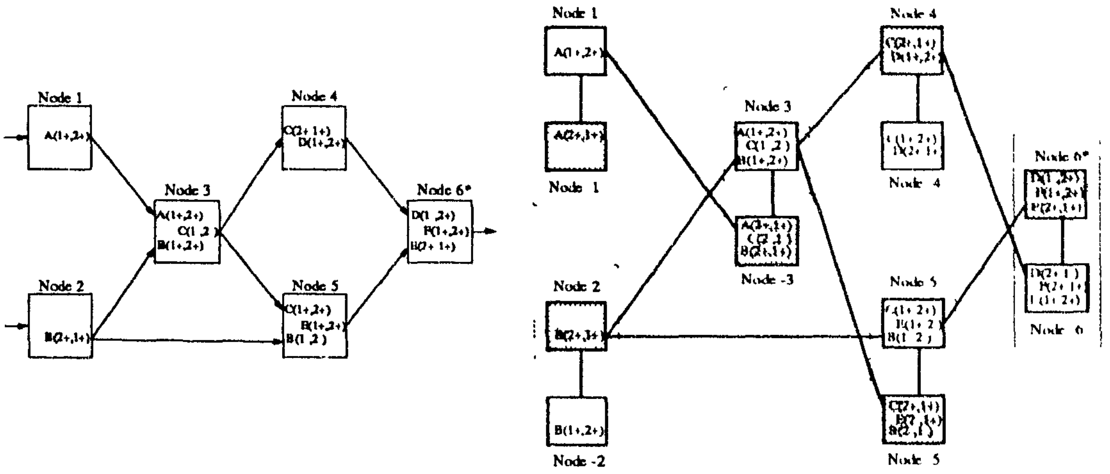
```
          DO 10 I = 1, N
             DO 10 J = 1, N
10              A(I,J) = I * J
          DO 20 I = 1, N
             DO 20 J = 1, N
20              B(J,I) = N - I
          DO 30 I = 1, N
             DO 30 J = 1, N
30              C(N-I+1,N-J+1) = A(I,J) + B(I,J)
          DO 40 I = 1, N
             DO 40 J = 1, N
40              D(I,J) = C(J,I) + 99
          DO 50 I = 1, N
             DO 50 J = 1, N
50              E(I,J) = C(I,J) + B(N-I+1,N-J+1)
          DO 60 I = 1, N
             DO 60 J = 1, N
60              F(I,J) = F(I-1,J) + D(N-I+1,J) + E(J,I)
```

Figure 2.5: A Collection of Typical Two-Dimensional FORTRAN Loops

two sets of nodes: those consisting of shaded boxes and those without, only the first is permissible because Node 6 is constrained by the loop-carried dependence which affects array F. Thus this set of shaded nodes forms the LCG used during the second phase of analysis. The LCG which depicts the results of Phase-1 analysis is shown in Figure 2.7(a).

Since Phase-1 analysis uncovered a compatible arrangement of nestings among the entire cluster, the compiler can proceed to the next phase of analysis. This second phase and the one that follows together form the Case-2 situation described above. During Phase-2 analysis, focus is on finding a compatible set of loop directions for each of the outside loops of the code blocks represented by the nodes of the transformed LCG. Therefore, the complementary nodes of the IG in this case are a copy of the primary nodes from Figure 2.7(a), with the direction of the first element of each AV reversed. The IG reflecting this set of alternative sequences and the resulting two-coloring is shown in Figure 2 7(b) Fortunately, Phase-2 analysis is successful too, so the compiler is able to proceed on to the third and final phase of analysis.

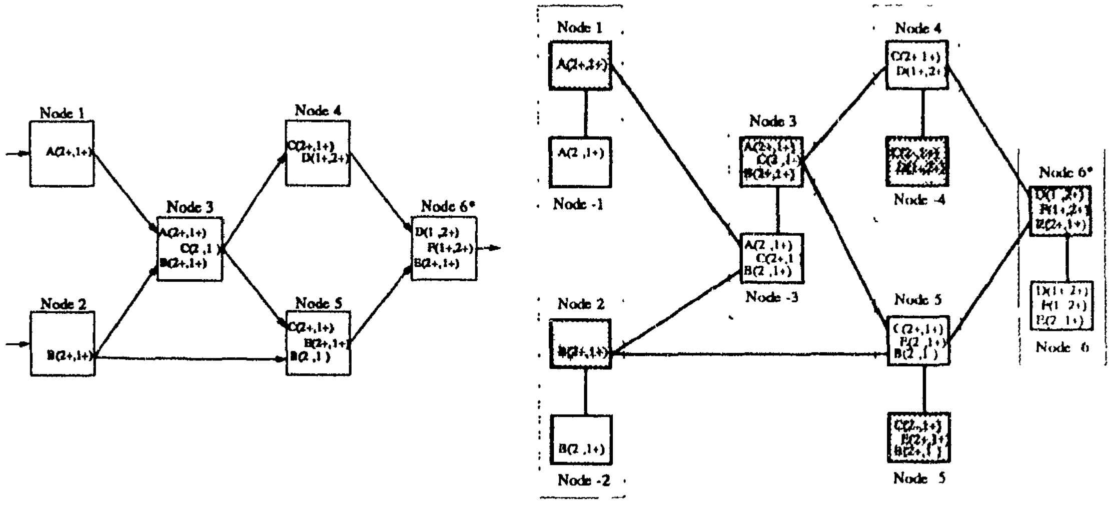
Phase 3 is just Phase 2 performed on the inside loop of each node in the cluster



(a) LCG prior to Interchange

(b) IG used for Case-1

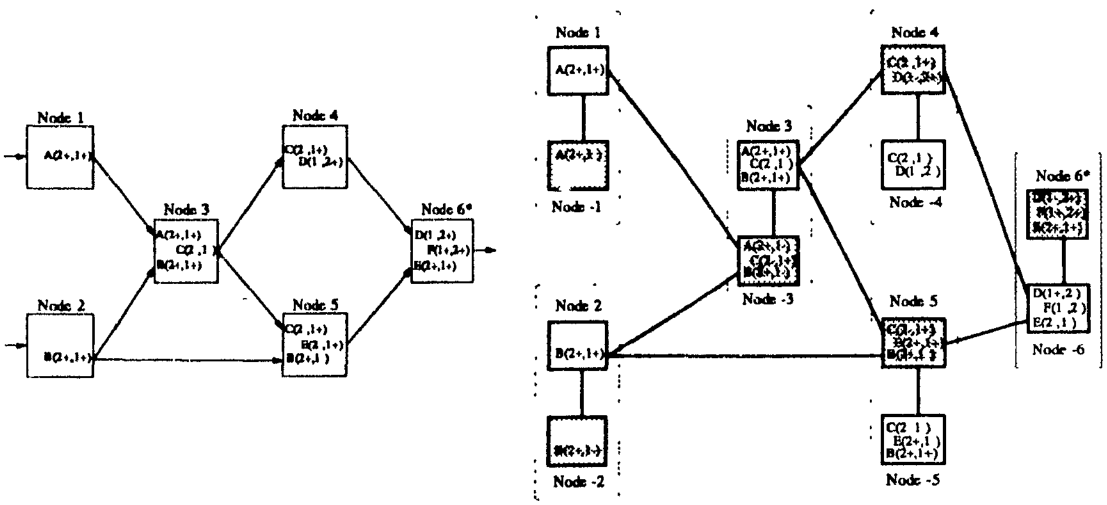
Figure 2.6: Phase-1 Analysis of a Two-Dimensional Loop



(a) LCG prior to Outer-Loop Reversal

(b) IG used for First Case-2 Transformation

Figure 2.7: Phase-2 Analysis of a Two-Dimensional Loop



(a) LCG prior to Inner-Loop Reversal

(b) IG used for Second Case-2 Transformation

Figure 2.8: Phase-3 Analysis of a Two-Dimensional Loop

rather than on the outside node. Using the output from the previous phase, i.e., the LCG shown in Figure 2.8(a), the compiler creates the IG shown in Figure 2.8(b). This time the complementary nodes are created by reversing the direction of the last element of each AV of a corresponding primary node. Phase-3 analysis is also successful, leaving the nodes in the shaded boxes in the IG in Figure 2.8(b) as the final set of nodes from the overall loop pipelining transformation. Again in this last phase, as in the previous one, the transformation required to bring the cluster into alignment is loop reversal.

Now that a compatible set of array reference orders is known, it is a straightforward process to fuse statements into a single code block and then eliminate the intermediate arrays by array contraction, as one done in the case of non-nested loops. To demonstrate this, we continue with the example, using the doubly-nested loops indicated by the shaded boxes in Figure 2.8(b). The retransformed code based upon the preceding analysis is shown at the top of the next page.

After Direction Reversal

```
      DO 10 I = 1, N
         DO 10 J = 1, N
10          A(J,N-I+1) = (N-I+1) * J
      DO 20 I = 1, N
         DO 20 J = 1, N
20          B(J,N-I+1) = N - (N-I+1)
      DO 30 I = 1, N
         DO 30 J = 1, N
30          C(N-J+1,I) = A(J,N-I+1) + B(J,N-I+1)
      DO 40 I = 1, N
         DO 40 J = 1, N
40          D(N-I+1,J) = C(N-J+1,I) + 99
      DO 50 I = 1, N
         DO 50 J = 1, N
50          E(J,I) = C(N-J+1,I) + B(J,N-I+1)
      DO 60 I = 1, N
         DO 60 J = 1, N
60          F(I,J) = F(I-1,J) + D(N-I+1,J) + E(J,I)
```

When loop fusion is applied, the original loop cluster is transformed into the single doubly-nested loop shown below, to the left, and once this is done, all of the array references except the last (the reference to the output array F) are replaced by corresponding scalar variables, as shown on the right.

After Loop Fusion

```
DO I = 1, N
   DO J = 1, N
      A(J,N-I+1) = (N-I+1) * J
      B(J,N-I+1) = 1 - I
      C(N-J+1,I) = A(J,N-I+1) + B(J,N-I+1)
      D(N-I+1,J) = C(N-J+1,I) + 99
      E(J,I) = C(N-J+1,I) + B(J,N-I+1)
      F(I,J) = F(I-1,J) + D(N-I+1,J) + E(J,I)
   END DO
END DO
```

After Array Contraction

```
DO I = 1, N
   DO J = 1, N
      a = (N-I+1) * J
      b = 1 - I
      c = a + b
      d = c + 99
      e = c + b
      F(I,J) = F(I-1,J) + d + e
   END DO
END DO
```

Note that in this case, the savings in storage and memory bandwidth is $5 \cdot n^2$, where n is the number of loops in the cluster. Again, in addition there are other advantages derived from increasing code-block size, advantages cited earlier for the non-nested case (see Section 2.4).

2.6 Analysis of k-Nested Loops

When the notions of the previous section are extended to the multiple nestings, the problem quickly becomes intractable because of the rapid growth of possibilities. Nonetheless, one special case does warrant mentioning; this is the extension of Case 2 from the previous section to the general case in which nesting of loops among the various code blocks comprising the cluster of loops are already compatible in the sense of our Case-1 transformation above. In other words, loop interchange is unnecessary to make the cluster compatible. In this instance, only the direction of the AVs in the various code blocks are allowed to vary. In this special case, analysis proceeds exactly as it did for Case 2 with a separate phase of analysis for each element of the AVs within the cluster.

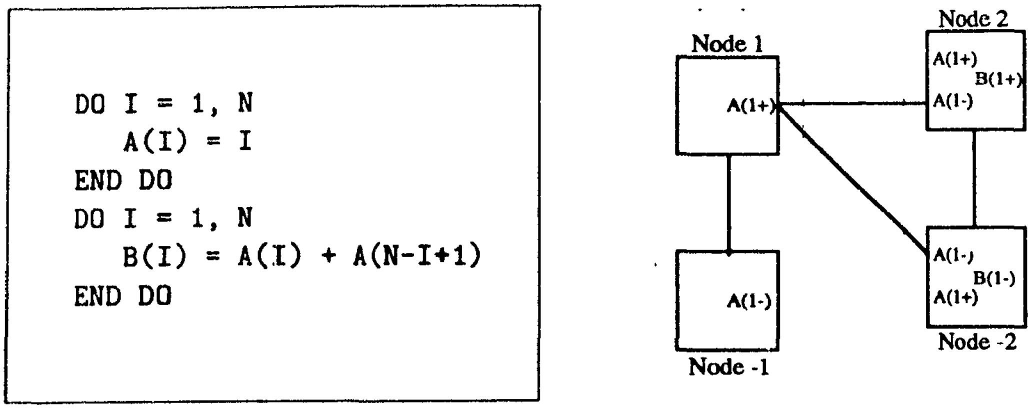
Chapter 3

Non-Compatible Loop Clusters

Not all loop clusters are able to be transformed into compatible clusters by the graph-coloring technique described in the last chapter. Nevertheless, we can usually partition a cluster into a set of smaller clusters which are compatible. Moreover, by cluster partitioning, much of the benefit from collective loop analysis can still be obtained. This then is the problem we examine in this chapter: how to partition a cluster of non-compatible loops efficiently to achieve best performance

We begin with analysis of a few simple loop clusters consisting of non-nested code blocks for which there exists no overall compatibility. From these example clusters we are able to see that the least expensive partitioning of non-compatible loops often occurs at forks in the loop communication graph (LCG). Moreover, we observe that these forks are always a part of odd-length cycles in the corresponding interference graph (IG). Odd-length cycles, however, are not the only cause of loop non-compatibility; a second cause is certain configurations of loops which are constrained by loop-carried dependence. This type of non-compatibility arises in situations in which there are two or more loops, each with a loop-carried dependence, connected by a *non-compatible chain* of loops. A chain is non-compatible if its constrained end nodes within the chain are separated by an odd-length distance from one another and the two nodes are of opposite color.

Based upon our analysis of the above non-compatibility factors, we are able to develop a heuristic algorithm to efficiently partition a cluster of non-compatible loops. We then formally describe the algorithm, and through a series of small FORTRAN examples, we illustrate its use. Finally we justify our approach and discuss its computational complexity.



```
DO I = 1, N
   A(I) = I
END DO
DO I = 1, N
   B(I) = A(I) + A(N-I+1)
END DO
```

(a) Non-Compatible Loop Directions among Two Loops

(b) An Interference Graph Which Is Not 2-Colorable

Figure 3.1: Two Non-Compatible Loops

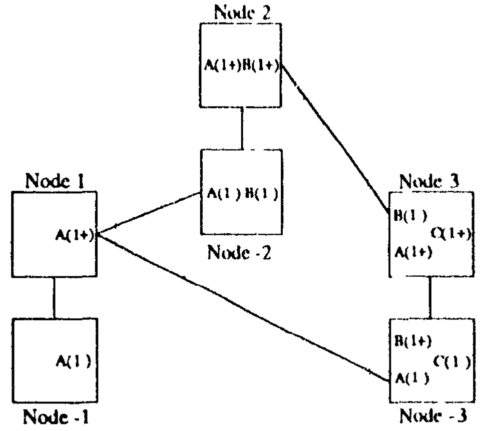
3.1 Sources of Non-Compatibility

In this section we examine the implications of loop non-compatibility. As noted, compatibility does not always exist among the constituent loops of a cluster. Therefore, as a first step, we would like to be able to identify the conditions which are at the source of the compatibility problem. As we will show, this information is important to being able to partition loops efficiently, both from the standpoint of the transformation itself, as well as from the standpoint of the performance of the resulting code. We begin our examination of non-compatibility analysis by illustrating two of the simplest cases. The first is shown in Figure 3.1. In this example, array A is generated in the first code block and referenced twice in the body of the second, each set of references being made in a different sequence, due to a different loop direction—the first set of references, in ascending order, and the second, in descending order. Observe that, in Figure 3.1(b), the IG is not two-colorable, thus providing graphic evidence of the non-compatibility of the two loops. Also, observe that the non-compatibility originates at the location of a *fork* in the LCG. For ease of expressibility, we refer to such locations as forks in the IG, as well.

Next let us complicate the situation by adding a third code block to a cluster, as shown in Figure 3 2. Notice that in Figure 3.2(b) the genciation of array A in Block

```
DO I = 1, N
    A(I) = I
END DO
DO I = 1,N
    B(I) = A(I)
END DO
DO I = 1, N
    C(I) = A(I) + B(N-I+1)
END DO
```

(a) Non-Compatible Directions among Multiple Loops



(b) An Interference Graph Which Is Not 2-Colorable

Figure 3.2: A Cluster of Three Non-Compatible Loops

1 constrains the allowable reference sequences of A in Blocks 2 and 3. Consequently, for the last two code blocks to become compatible with the first, array A must be generated in both ascending and descending sequences simultaneously, an impossibility. Notice again, our previous observation once again applies: non compatibility originates at the location of a fork—also notice that the graph is not two colorable. The fact that neither this graph nor the previous one is two-colorable, hence, the fact that we are unable to find a compatible set of loop directions, is readily explained by a fundamental theorem from graph theory [BM76, Deo74, Har69]:

Theorem 3.1 *A graph with at least one edge is two-chromatic if and only if it has no cycles of odd length.*

proof:

if: Assume G is two-chromatic; its node set V might then be partitioned into two sets V_1 and V_2 so that every edge of G joins a point of V_1 and V_2. Therefore, every cycle $v_1, v_2, v_3, \ldots, v_n, v_1$ in G necessarily has its nodes with odd subscripts in V_1 and nodes with even subscripts in V_2. As a result, the length of every cycle is even.

only if: Assume, without loss of generality, that G is connected (otherwise the components would be considered separately). Take any node $v_1 \in V$ and let V_1 consist of

v_1 and all points at even distance from v_1, while $V_2 = V - V_1$. Since all the cycles of G are even, every line of G joins a node of V_1 with a node of V_2. Now suppose there is a line uv that joins two nodes of V_1. The union of shortest paths from v_1 to v and from v_1 to u together with the line uv contains ar odd cycle, a contradiction. □

Obviously the odd cycle in the IG in Figures 3.1 and 3.2 must be eliminated without alteing the meaning of the original code, if transformation is to be possible. Unfortunately, non-compatibilities which are inherent in the original code cannot be eliminated by further direction transformations. Even so, their adverse performance effects can be partially alleviated by breaking the original non-compatible cluster into smaller clusters, each of which is compatible. The loops in each of these subclusters can then be fused and transformations, such as array contraction, performed on the resulting fused loops, as described in the previous chapter (see Section 2.4, page 20).

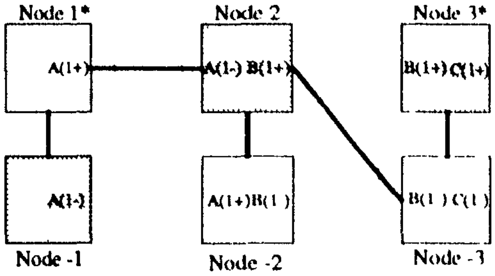
There is one additional source of compatibility we have yet to discuss, and that is the non-compatibility which sometimes occurs among loops in a cluster as a result of conflicting reference-order constraints caused by loop-carried dependence. An example of just such a situation is shown in Figure 3.3. Here the source of loop non-compatibility is the conflicting array-reference orders imposed on the first and third loops by the loop-carried dependence within each loop, cf. Figure 3.3(a). In Figure 3.3(b) we show the corresponding IG as it would appear following compatibility analysis. Notice that in this graph the two non-compatible nodes, Nodes 1 and 3, are separated by a non-compatible (odd-length) chain of nodes. From our previous analysis we know this chain is non-compatible because the two end nodes in the IG (Nodes 1 and 3) are of opposite color. All that would be needed to make this chain compatible would be to remove the loop-carried dependence from either of the two end nodes—but of course, there is no way of doing this. Consequently, there is no remedy but to remove an edge from the chain, in effect, separating the constrained nodes by putting them in different partitions, as in the previous situations. There are usually several alternatives for accomplishing such a partitioning, and after we have laid additional groundwork, we return to this problem to see how partitioning might be accomplished.

3.1.1 Properties of Non-Compatible Clusters

In this section we continue our analysis of non-compatible loops. In the process we isolate additional properties, besides those already mentioned, which affect partitioning effectiveness. In the course of this investigation we establish a number of

```
DO I = 1, N
     A(I) = A(I-1) + 10
END DO
DO I = 1,N
     B(I) = A(N-I+1)
END DO
DO I = 1, N
     C(I) = C(I-1) + B(N-I+1)
END DO
```

(a) Non-Compatible Loop-Carried Dependence



(b) An Interference Graph Which Is Not 2-Colorable

Figure 3.3: A Non-Compatible Chain of Loops

guidelines which ultimately form the basis of the partitioning algorithm we later describe. Before we begin, however, a comment about the graphical construct we use during partitioning analysis: the Spanning Graph (SG)

A spanning graph is the two-colored spanning tree induced from the IG, with compatible edges from the IG added as arcs and non-compatible edges marked as X-edges. During analysis, the SG is decorated with other information used during the partitioning process; details of this information are explained as we go along. Because the SG starts out as an IG, we periodically refer to the two graphs interchangeably, choosing one name or the other depending upon the graph property we wish to emphasize. As a means of distinction, however, we consistently refer to undirected "edges" in the IG, as opposed to directed "arcs" in the SG.

There are two reasons for using the IG for loop partitioning: First, since it will already have been constructed during previous graph-coloring analysis, construction is free. And second, it contains the information needed to perform partitioning, organized in the way that can be efficiently accessed. Of course, this second consideration is most important.

Without question, the primary objective of cluster partitioning is to aggregate clusters of loops in such a way so as to achieve the best performance overall, and we now show that the best way to accomplish this objective is to find a minimum cost edge-cut of the graph. To understand exactly what is implied by minimum cost, one need only examine Figure 3.2. Notice, as was proven above, non-compatibility occurs if there exists an odd-length cycle in the graph, cf., Figure 3.2(b). Consequently, to make the nodes in the graph compatible we must break this cycle, and it must be broken in at least two locations; otherwise, there would be no way to fuse the loops within the individual clusters. Furthermore, were we to cut the graph at only one location, there would be no valid corresponding program representation, as one might readily confirm.[1]

For the odd-length cycle in Figure 3.2(b) there are ten combinations of edge-cuts which will partition the cycle,

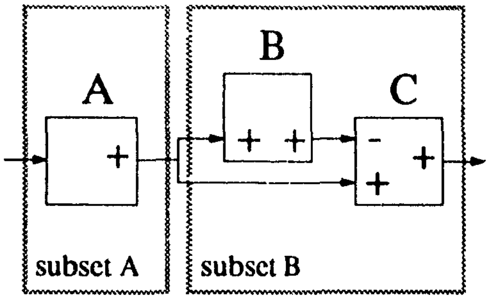
$$C(5,2) = \frac{5!}{3! \cdot 2!} = 10$$

howevei, not all of these combinations are valid, and others are more costly than the rest. For example, a cut between Node 2 and Node −2 would not be valid since putting these nodes in separate partitions has no meaning in the context of the actual loop cluster. Likewise, a partition such as $\{(1,-2),(2,3)\}$ would not be valid either, since the corresponding loop organization is not programmable. Consequently $2! \cdot 1! = 2$ combinations remain: $\{(1,-2),(1,-3)\}$ and $\{(2,3),(1,-3)\}$. Of these two alternatives, the first is least costly since it affects the fewest number of arrays overall, in this case, only array A; whereas the second edge-cut, $\{(2,3),(1,-3)\}$, affects two arrays, A and B.[2] Note that by cutting an edge we imply the need to pass corresponding array values between code blocks. If we partition the cluster using the first edge-cut, the result would be as shown in Figure 3.4. In Figure 3.4(a) we show the partitioned LCG, and in Figure 3.4(b) we show the resulting code after fusion. Although a simple case, this example introduces the intuition behind the partitioning strategy we describe next.

With this next example, we establish a general framework for solving the cluster partitioning problem, which we will build upon in later examples. The FORTRAN code for this problem, listed in Figure 3.5(a), is similar to the code in Chapter 2 used to verify compatibility of non-nested loops (cf., Figure 2.1, page 12), except the loops in this instance are non-compatible. The non-compatibility of these loops would be

[1] This is not true in all cases, e g , certain types of multiprocessor programming, as we will explain in Chapter 4 It does however apply in all cases of uniprocessor programming—which is the focus of our research

[2] One might easily verify that the same observations apply with regard to the simplest example, shown in Figure 3 1, page 31



```
DO I = 1, N
   A(I) = I
END DO
DO I = 1,N
   B(N-I+1) = A(N-I+1)
   C(I) = A(I) + B(N-I+1)
END DO
```

(a) The Loop Communication Graph **(b) Code after Loop Fusion**

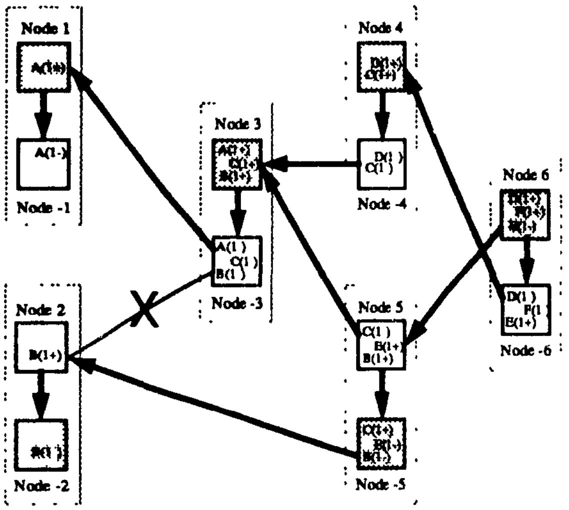
Figure 3.4: A Partition of Three Non Compatible Loops

discovered during initial graph-coloring analysis. Recall that, during this analysis, a breadth-first spanning tree is constructed and colored, as shown by the bold edges and shaded nodes in the IG, in Figure 3.5(b). As a part of the process, remaining edges, i.e., those that are not already part of the tree, are checked, and any that can be added, without violating the two-coloring constraint, are added. It is at this point the IG becomes a Spanning Graph, or SG. Arc (−4,3) is added to the SG because the nodes incident to it are each of a different color. On the other hand, the nodes incident to edge (2,−3) in the IG are of the same color and so an arc would not be added. Note we indicate the non-compatibility of edge (2,−3) by an X

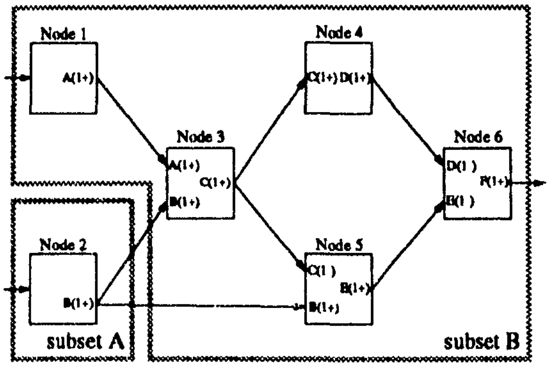
Now suppose we were to add the non-compatible edge (2,−3) to the tree, knowing that this is not allowed. The result would be the creation of the odd-length cycle $\{2,-3,3,5,-5,2\}$. Since this is the cycle preventing compatibility, it is the one that must be partitioned. Here we consider only *fundamental odd-length cycles* in our analysis, because unless all such cycles are broken, non-compatibility will remain. By fundamental cycles we mean *chordless cycles*, therefore, the cycle $\{2,-3,3,-4,4,-6,6,5,-5,2\}$ is not a fundamental cycle because of chord (3,5) Before continuing, let us reemphasize the following points: *1)* at least two edges of the odd-length cycle must be cut to effectively partition the cycle, *2)* the edge cut must cut both chains extending between the fork and join of the cycle, and *3)* the least costly edges to cut are those which correspond to a fork in the LCG, since usually only a single array is affected If we apply these notions to this particular problem in

```
      DO 10 I = 1, N
10       A(I) = I
      DO 20 I = 1, N
20       B(I) = N - I
      DO 30 I = 1, N
30       C(I) = A(I) + B(I)
      DO 40 I = 1, N
40       D(I) = C(I) + 99
      DO 50 I = 1, N
50       E(I) = C(N-I+1) + B(I)
      DO 60 I = 1, N
60       F(I) = D(I) + E(N-I+1)
```

(a) A Non-Compatible Cluster of Loops



(b) The Spanning Graph (SG) Induced from the IG



(c) The Partitioned LCG

```
      DO 10 I = 1, N
         B(I) = N - I
10    CONTINUE
      DO 20 I = 1, N
         a = I
         c = a + B(I)
         d = c + 99
         e = c + B(N-I+1)
         F(I) = d + e
20    CONTINUE
```

(d) Code after Loop Fusion and Array Contraction

Figure 3.5: A More Complex Cluster of Non-Compatible Loops

Figure 3.5, we once more find we are able to partition the graph, just as effectively as before (see Figure 3.5(c)). Moreover, we are able to obtain much the same benefit we were able to obtain from transforming a fully compatible cluster, as is evident from the transformed code in Figure 3.5(d). Furthermore, this benefit is achieved with only slightly additional analysis.

There is one additional assumption we make with respect to the presence of even-length cycles in the SG, e.g., the cycle $\{3,-4,4,-6,6,5,3\}$ in Figure 3.5(b). One might rightly ask: How does the addition of these cycles affect the overall analysis? The answer to this question lies in the following theorem:

Theorem 3.2 *The graph formed by inserting an edge between any two nodes of a two-colored graph will itself be two-colored if and only if the nodes incident to the additional edge are opposite in color.*

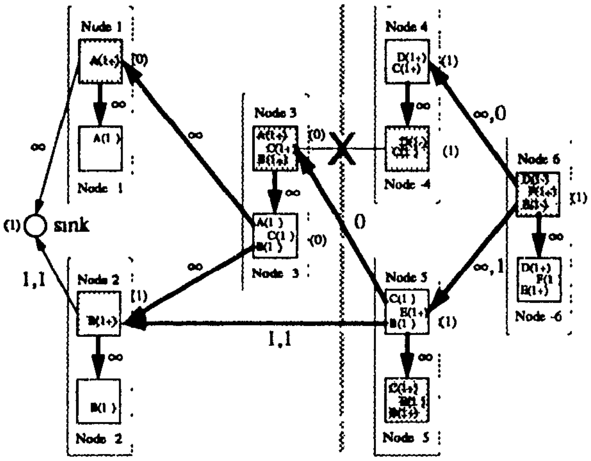
Proof of the theorem is immediate, by construction. Theorem 3.2 assures us that the even-length cycles have no adverse effect, as far as our analysis is concerned. We might add, however, not only is it safe to insert the compatible edges into the graph, it is essential to do so to ensure completeness. Otherwise, without these edges we would be unable to apply the global analysis needed to obtain minimum cost partitioning in more complex situations. The significance of this last statement will become clear as our analysis continues.

So far we have only encountered simple cases in which there was little interaction between an odd-length cycle and the rest of the graph. Unfortunately, the situation is often more complicated, and cutting just a single arc, as we did so far, is not enough to effect complete partitioning. This is case in our next example, shown in Figure 3.6. As in earlier examples, the original code is in Figure 3.6(a), and the corresponding SG is in Figure 3.6(b). Also as with previous examples, we show in Figure 3.6(b) the breadth-first spanning tree and node coloring, constructed as previously described. Notice, however, this time the odd-length cycle and the even-length cycle have been switched. The odd-length cycle is now the cycle $\{3,-4,4,6,6,3\}$. As a consequence, the previous strategy to partition the graph no longer works. The reason is that there is now an extra edge that must be dealt with that is outside of the odd length cycle, edge $(2,5)$.

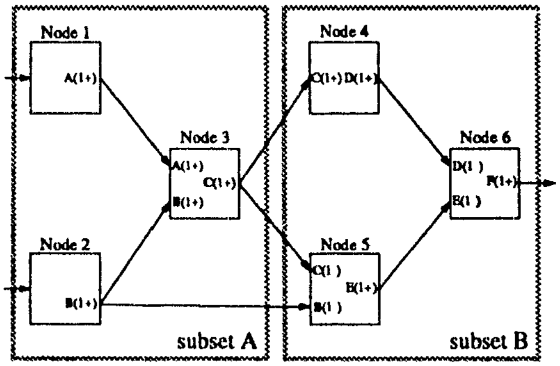
From our previous examples we know that edges $(3,4)$ and $(3,5)$ must be cut to effect the most efficient edge-cut of the odd-length cycle. It is also clear from the figure that edge $(2,5)$ must be cut; otherwise, the loop cluster will not be separated.

```
      DO 10 I = 1, N
10       A(I) = I
      DO 20 I = 1, N
20       B(I) = N - I
      DO 30 I = 1, N
30       C(I) = A(I) + B(I)
      DO 40 I = 1, N
40       D(I) = C(I) + 99
      DO 50 I = 1, N
50       E(I) = C(N-I+1) + B(N-I+1)
      DO 60 I = 1, N
60       F(I) = D(N-I+1) + E(N-I+1)
```

(a) A Non-Compatible Cluster of Loops



(b) Flow Analysis of the Spanning Graph



c. The Partitioned LCG

```
      DO 10 I = 1, N
         a = I
         B(I) = N - I
         C(I) = a + B(I)
10    CONTINUE
      DO 20 I = 1, N
         d = C(N-I+1) + 99
         e = C(I) + B(I)
         F(I) = d + e
20    CONTINUE
```

(d) Code after Loop Fusion and Array Contraction

Figure 3.6: Another Case of Non-Compatible Loops

As a further condition, each cluster created as a result of the partition must itself be a *fusionable partition*. By a fusionable partition we mean that the loops comprising the partition must be compatible in order that they be later combined into a single loop, by loop fusion. Furthermore, we would like the edge-cut that accomplishes the partitioning to be least cost in terms of the number of edges cut. To achieve these objectives, we expand our previous methodology to incorporate a graph partitioning technique based upon the well-known *Max-Flow Min-Cut* theorem [Law76].

Theorem 3.3 (Max-Flow Min-Cut) *The maximum value of an (s,t) flow is equal to the minimum capacity (s,t)-cutset.*

The s in the above theorem represents the source of flow, and the t, the sink. For the moment, one might view the "flow" as the pipelined transfer of array elements from one code block to the next.[3] Accordingly, an (s,t) flow is just the flow between the source and the sink, and an (s,t)-cutset is any partition of the network that separates the nodes into two sets, one set containing the source and the other, the sink.

To find the max-flow and min-cut through a network, a series of *flow-augmenting paths* are successively added together until the network becomes saturated. At the point of saturation, network flow becomes maximal, and as Theorem 3.3 suggests, the flow at this point is equal to the flow across the minimum capacity cutset. This property is formally stated in the following related theorem.

Theorem 3.4 *A flow is maximal if and only if it admits no augmenting path from s to t.*

The above theorem assumes several basic network properties apply, based upon the so-called *conservation law.* The conservation law states that the flow into a node must equal the flow leaving it, or

$$\sum_j x_{ji} = \sum_j x_{ij} \qquad (3.1)$$

In the above equation x_{ij} is the flow through arc (i,j). When the preceding relationship is extended to the entire network we have

$$\sum_j x_{ji} - \sum_j x_{ij} = \begin{cases} -v, i = s \\ 0, i \neq s, t \\ v, i = t \end{cases} \qquad (3.2)$$

[3] Later we will change this definition slightly to suit our specific purpose.

where v is the value of the flow. Besides flow conservation, on each arc we also have a *capacity constraint*:

$$0 \leq x_{ij} \leq c_{ij} \tag{3.3}$$

c_{ij} is the flow capacity of the arc. Any flow satisfying condition 3.2 and condition 3.3 is called a *feasible flow*. Lastly, if P is an undirected path from s to t, an arc $(i,j) \in P$ is a *forward* arc if it is directed from s to t; otherwise, it is a *backward* arc. P is said to be a flow-augmenting path with respect to flow x_{ij} if $x_{ij} < c_{ij}$ for each forward arc, and $x_{ij} > 0$ for each backward arc. When we apply these notions, the solution technique becomes straight forward: Saturate the network with augmenting flows in order to find a minimum capacity cutset. This cutset, in turn, represents the least-cost partition of the SG that we are seeking.

The initial phase in the partitioning process is to transform the SG in Figure 3.6(b) into the structure of a flow network, as specified in preceding flow-problem formulation. To do this, we make the following correspondence between the two problems:

1. We use the SG to represent the network, as shown in Figure 3.6(b).

2. We assume that flows go against the flow-dependence arcs in the IG, i.e., instead of the flow going from, say, Node 2 to Node 5, we assume it flows in the opposite direction, from Node 5 to Node 2, as shown in the figure.

3. Because the flows are reversed, we reverse the network source and sink; for example, we let Node 6 be our source, instead of a sink, and in situations in which there are multiple sources and/or sinks, we extend the network by adding an artificial source or sink, as shown. In the Figure 3.6(b) the artificial sink has been added to the SG to connect the input nodes, 1 and 2. Extending the graph, as we have done, is the usual way to handle multiple sources and sinks; however, instead of assigning infinite capacity to these pseudo-arcs as customarily done, we sometimes assign a lesser capacity, as described next.

4. We assign a zero capacity to each input arc of a join in the SG (or conversely, to each fork in the IG) if the input arc of the SG is adjacent to an X-edge in the IG. For example IG edge $(3, -4)$ is marked with an X, so the adjacent SG input arc $(5, 3)$ is given a capacity of zero.

5. Next we associate with each node a temporary variable, called *node_capacity*, and for each output node of the IG, we initialize the value of this variable to 1;

each remaining node capacity is initialized to zero. Then at each terminal node encountered during the breadth-first traversal of the IG (used to create the SG), we add the capacity of the origin node to the current node's capacity, provided the corresponding edge is neither an X-edge or an adjacent zero-weighted arc. At each "compatible" join encountered, i.e., for each join node in the SG that does not have an X-edge or zero-weighted input arc, we assign the arc's origin node capacity to the corresponding arc, if the corresponding origin node is non zero. In Figure 3.6(b) for example, Node 3 has zero capacity, Node 2, a capacity of 1 (taken from Node 5); and the sink node, a capacity of 1 also (taken from Node 2).

6. And lastly, we add infinite capacities to all remaining arcs, to ensure that none are cut.

Item 4 is done to ensure that every odd-length cycle is cut, although other higher priority cuts can, and often do, occur closer to the root(s) of the SG which supersede these "later" cuts. As a result, these later cuts might not actually ever be effected. Ultimately it is the flow-augmenting path algorithm that determines whether these cuts are actually used to partition the graph.

The motivation for Item 5 might be unclear at this point; however, we explain the rational behind it after this next example. For the moment, the underlying notion is merely that the capacities must reflect the relative priority with which cuts are to be made.

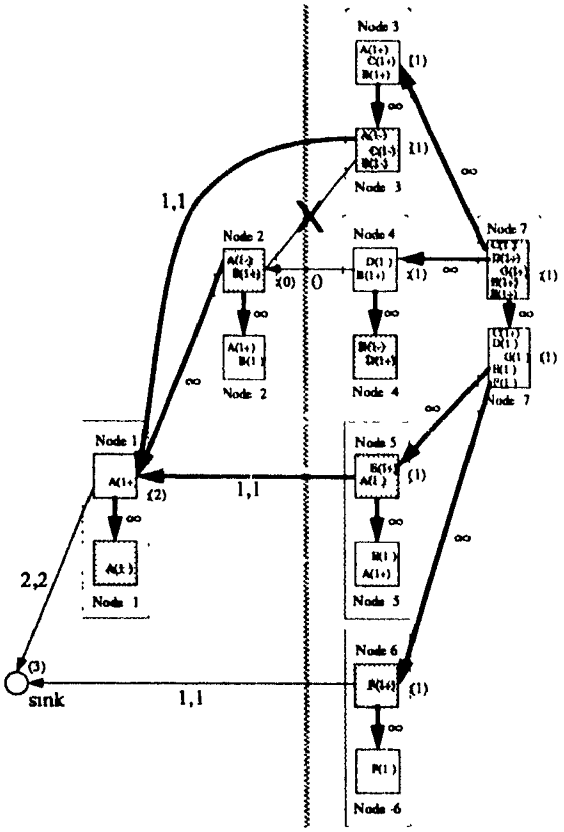
Let us return to the problem shown in Figure 3.6 to see how the above strategy is applied. Either during the graph-coloring process or immediately following it, pseudo source nodes and/or pseudo sink nodes are added to the graph, if either are necessary. In this instance, since the SG contains two leaf blocks, Blocks 1 and 2, a sink node is created with arcs extending from the primary nodes of these blocks to this sink.

The compiler next assigns arc capacities, as described in Step 5. For the purpose of clarity, we describe how these capacities are derived rather than enumerating the steps taken during their derivation; the exact procedure is described in the next section. Edge (3,–4) from the IG in Figure 3.6 is an X-edge and therefore, it is not included in the SG; arc (5,3), the adjacent arc at the join in the IG at Node 3 is assigned a zero capacity. The second join at Node 2 is part of the even-length fundamental cycle between Blocks 2, 3, and 5. The capacity of Node –3 is zero, because of the zero capacity at Node 3, hence, it has no effect on the capacity of Node 2. On the other hand, Node 5 receives a unit capacity from Node 6, which increments the capacity of Node 2, and accordingly, arc (5,2) receives a capacity of 1. Applying the same rational to the sink node, we derive a capacity on the arc (2, **sink**) of 1 also All remaining arcs in the graph are then given an infinite capacity.

Once these capacities have been assigned, the SG is ready for network flow analysis, and the flow-augmenting path algorithm is applied. In this instance, only one path is open, and it is saturated after only one flow-augmenting path is found, $\{6, -6, 5, 2, sink\}$. At the point of network saturation the graph becomes effectively partitioned, as shown. Although we are assured of the compatibility of nodes prior to the cut, we cannot be assured of the compatibility of nodes after it, since a single cut is often insufficient to break all odd-length cycles in the graph. Therefore, graph-coloring is applied once more to the remaining (left-most) subgraph. In this instance the subgraph remains unchanged by the ensuing analysis, so partitioning is complete. The LCG, after partitioning, is shown in Figure 3.6(c). Since the compiler now knows that each of the two partitions are compatible, the corresponding loops in each partition can be fused and array contraction applied, as shown in Figure 3.6(d). As can be seen from the transformed code, the benefit of partitioning can be significant; in this case, it results in elimination of loop control overhead for three loops and scalar-variable replacement of three of the original six arrays.

Before proceeding with the formal description of our loop partitioning algorithm, let us return to the issue of the necessity for Item 5 of the network transformation description (on page 41). Item 5 gives criteria for assigning arc capacities based upon cumulative earlier capacities. The reason for establishing capacities as prescribed by this item is two-fold: *1)* to ensure sufficient flow reaches all arcs in the graph, especially those incident to distant joins, and *2)* to ensure that arcs incident to joins in the SG are given priority over cuts along the respective paths to these joins. The first reason is based upon the fact that, in the absence of sufficient flow, lower-priority arcs can be starved of capacity (and hence cut) preventing flow from reaching possible distant cuts of higher-priority. On the other hand, the arc capacities themselves serve as a means of establishing priority, that is, a means of differentiating the relative importance of one cut as opposed to another.

The impact of capacity assignments is illustrated in the partitioning problem shown in Figure 3 7 Figure 3.7(a) depicts the SG generated from the FORTRAN source code shown in Figure 3.7(b). There are two situations present in this code which differentiate it from our previous examples: *1)* adjacent cycles, and *2)* loosely



(a) Partitioned Spanning Graph

```
      DO 10 I=1, N
10        A(I) = 36
      DO 20 I=1, N
20        B(I) = A(N-I+1) + 99
      DO 30 I=1, N
30        C(I) = A(I) + B(I)
      DO 40 I=1, N
40        D(N-I+1) = B(I) * 8 + 2
      DO 50 I=1, N
50        E(I) = A(N-I+1) / 4
      DO 60 I=1, N
60        F(I) = N
      DO 70 I=1, N
70        G(I) = C(N-I+1) * D(I) + E(I) * F(I)
```

(b) Original FORTRAN Code

```
      DO 10 I=1, N
          A(N-I+I) = 36
10        B(I) = A(N-I+1) + 99
      DO 20 I=1, N
          c = A(N-I+1) + B(N-I+1)
          d = B(I) * 8 + 2
          e = A(N-I+1) / 4
          f = N
20        G(I) = c * d + e * f
```

(c) Optimal Solution

Figure 3.7: Optimal Partitioning using Node Capacities

connected nodes [4]

The adjacent cycles are those depicted in the upper-left portion of the figure, $\{1,2,-3\}$, $\{2,-3,3,7,4\}$, and $\{1,2,4,7,-7,5\}$. The primary characteristic of such cycles is that the fork and join of one cycle is somehow interleaved with the fork and join of another. In situations of this type, it is necessaiy to ensure sufficient flow reaches the farthest cycle so that should this cycle need to be cut, it will be cut at its least-cost location—the location corresponding to a fork in the LCG. In this instance, since cycle $\{2,-3,3,7,4\}$ is odd-length, it must be cut, but doing so forces the later cycles, $\{1,2,-3\}$, and $\{1,2,4,7,-7,5\}$, to also be cut, even though these later cycles are both of even length. Notice how the node capacities ensure that all of the cuts occur at the correct location. Notice too that this property holds, regardless of whether other subgraphs of intervening chains or even-length cycles are added to these cycles, for example, imagine additional nodes being inserted along arc $(-3,1)$.

Loosely connected nodes are those which are not a part of some outer-level cycle, such as Node 6. Again, as in the previous situation, node capacity is required to ensure cuts are made at the correct location. Specifically, had node capacities not been used, arc $\{-7,6\}$ would have had the same capacity as arc $\{6,\textbf{sink}\}$, and as a result, the flow-augmenting path algorithm would have incorrectly cut arc $\{-7,6\}$, isolating Node 6, instead of making the least-cost cut of the arc to the pseudo sink, as shown. In both this instance and the previous one, adherence to the capacity-weighting scheme described in Item 5, led to optimal partitioning of the graph; whereas, had capacities been assigned less discriminately, optimality would not have been achieved.

3.1.2 The Partitioning Method and Refinements

In the last section we provided an intuitive description of how loop-cluster partitioning might be accomplished In this section we formalize the method and provide additional examples illustrating the application of cluster partitioning to special cases involving multiple partitions and then constrained loops. First, we begin with a formal description of the algorithm:

Algorithm III *Partition a cluster of non-compatible loops.*

[4] We are more precise with these terms in Section 3 2

input: An interference graph, IG $= [[N, \overline{N}], E]$, constructed using Algorithm 1 (see page 15).

A colorset$_c$ = {color$_1$, color$_2$} for each output node in the LCG,

A color_preference which determines the color of the set to be chosen in the event the loop cluster contains constrained nodes, initially color_preference is undefined,

A checkset which contains the current nodes being visited during breadth-first traversal, and a nextset which contains the next level of nodes to visit,

A set of noncompatibles = $\emptyset$, which contains any edges identified as having a non-compatible array-reference order with that of its neighbor node; we refer to these edges as X-edges,

A global Boolean value constrained, initialized to false, indicating non compatibility because of loop-carried dependence constraints, and

A Boolean return value, compatible, with an initial value of true.

output: The IG partitioned into components, each component representing a compatible loop cluster.

procedure:

```
FUNCTION colorTree (checkset): compatible
FORALL $n \in$ checkset
    FORALL input arcs $X_{nm}$ and the arc to complement node $\overline{n}$
        IF the terminus $m$ is not colored
            color $m$ the opposite color of $n$
            set $m$.node_capacity = 0
        IF $m$.colorset $\neq$ $n$.colorset
        ;; establish a colorset correspondence
            alias $m$.colorset and $n$.colorset
        IF $m$ is the opposite color of $n$
        ;; set node and arc capacities
            increment $m$ node_capacity by $n$.node_capacity
            IF $m$ has multiple output ports
                $X_{nm}$.arc_capacity = $n$ node_capacity
            ELSE
                $X_{nm}$.arc_capacity = $\infty$
            add $m$ to nextset
        ELSE IF $m$ is the same color as $n$
        ;; cut the non-compatible edge
            $m$.node_capacity = $X_{nm}$.arc_capacity = 0
            compatible = false
        IF $m$ is constrained ;; i.e., it contains a loop-carried dependence
            IF color_preference has not been set
                set color_preference to the color of $m$
            ELSE IF the color of $m \neq$ color_preference
                constrained = true
    IF the nextset is not empty
        compatible = colorTree(nextset)
RETURN compatible


FUNCTION partition (graph):  compatible
;; associate a separate colorset with each output node
    source_nodes = getSources(graph)
    FORALL $n \in$ source_nodes
        $n$.colorset = colorset$_n$
;; color the graph
    compatible = colorTree(source_nodes)
IF NOT compatible
;; create a pseudo-source and pseudo-sink, if necessary
    IF numberOfSources(graph) = 1
        source = source_nodes
    ELSE
        create source
        FORALL $n \in$ source_nodes
            attach an arc from source to $n$
    sink_nodes = getSinks(graph)
    IF numberOfSinks(graph) > 1
        create sink
        FORALL $n \in$ sink_nodes
```

```
            attach an arc from $n$ to sink
            IF $n$.node_capacity > 0
                $X_{nsink}$.arc_capacity = $n$.node_capacity
    ;; partition the graph
        subgraph = flowAugmentingPath(source)
    RETURN partition(subgraph)
```

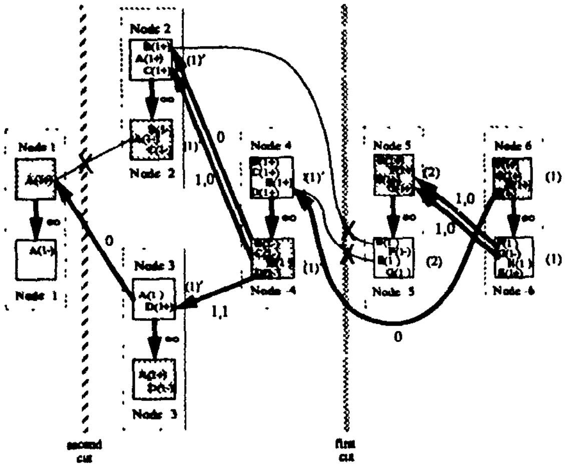
Notice that the colorTree function sets a global variable in the event of a loop carried dependence constraint. To partition the graph when such situations occur requires a modification to the algorithm, which we describe through an illustration later in this section. Notice too that as long as the tail-end subgraph of the SG is non compatible, the algorithm partitions the graph into increasingly smaller subgraphs. This is accomplished by recursive application of the partition function applied to the subgraph on the sink side the SG. As a result, an original cluster of loops might be partitioned several times, each partition resulting in at least one additional fusionable cluster of loops being found.

We examine these processes further in our next example, shown in Figure 3.8. Besides multiple partitioning, this example illustrates a couple of other unique situations which have not been addressed yet, specifically, the handling of multiple edges between nodes, and the treatment of adjacent cycles when multiple cuts are involved. The FORTRAN loop cluster used in this example is shown in Figure 3.8(a), prior to loop transformation, and the corresponding IG with its induced SG are shown in Figure 3.8(b). Examination of the figure reveals that the original loops are non compatible because there exists three odd-length cycles in the IG, the first involving Blocks 4–6; the second, Blocks 2, 4, and 5, and the third, Blocks 1–4.

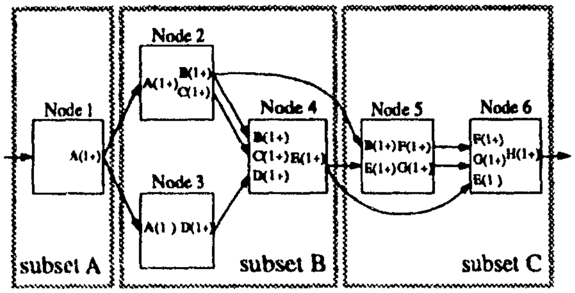
Notice in Figure 3.8(b), how we depict the situation in which multiple arrays are passed between code blocks, for example, arrays F and G being passed between Blocks 5 and 6. Each array is represented individually since a separate cost is associated with each. In this case, the capacity of Node 5 is 2, and the capacity of each connecting input arc 1, as shown. Next, notice that the arc adjacent to each X edge has its capacity set to zero, but only for those edges corresponding to the same array. For example, the capacity of arc $(6,4)$ for array E is zero, corresponding to X edge $(4,-5)$ for array E, and the capacity of one of the two adjacent arcs $(-4,2)$ is zero (for array B), but not the other (array C). Consequently, the capacity of Node 2 is 1, not 2. Next, notice the first cut in the SG is made without a single flow augmenting path

```
      DO 10 I=1, N
10       A(I) = I
      DO 20 I=1, N
         B(I) = A(I) * 2 + 3
20       C(I) = B(I) + 99
      DO 30 I=1, N
30       D(I) = A(N-I+1) + 6
      DO 40 I=1, N
40       E(I) = B(I) + C(I) * D(I)
      DO 50 I=1, N
         F(I) = B(I) * 4 + 2
50       G(I) = E(I) * 8 - 3
      DO 60 I=1, N
60       H(I) = F(I) + G(I) * E(N-I+1)
```

(a) A Non-Compatible Cluster of Loops



(b) The Corresponding SG Showing the Partitioning



(c) The Partitioned LCG

```
      DO 10 I=1, N
10       A(I) = I
      DO 20 I=1, N
         b = A(I) * 2 + 3
         B(I) = b
         c = b + 99
         d = A(N-I+1) + 6
20       E(I) = b + c * d
      DO 30 I=1, N
         f = B(I) * 4 + 2
         g = E(I) * 8 - 3
30       H(I) = f + g * E(N-I+1)
```

(d) Code after Loop Fusion and Array Contraction

Figure 3.8: Multiple-Cut Loop Partitioning

ever being made. Also notice that following the initial partitioning, the capacity of all adjacent arcs affected by the initial cut remains unchanged, since the cost of cutting the arc remains valid In other words, the zero capacity on arc $(-1,2)$ for array B is not reevaluated for the second iteration of the flow-augmenting path algorithm on subgraph $\{1,2,3,4\}$; whereas the remaining node and arc capacities are (In the figure the revised node capacities in the middle partition are denoted with a hash mark.). The second algorithm application cuts the graph a second time, as shown, resulting in two cuts and three subgraphs overall, cf Figure 3.8(c) Although the partitioning results consist of three fusionable partitions, instead of two, the benefit of transformation remains high In this case, five of the original eight arrays were still able to be eliminated, along with the loop overhead from three loops

So far we have not discussed cluster partitioning for the case of non compatible loop clusters caused by constrained loops that are separated an odd number of arcs from one another.[5] In situations of this type, a least-cost cut must be found between the two loops. However, since this type of non-compatibility can often be broken at the same time a cluster is being partitioned to remove the previous type of non compatibilities, we do not attempt to remove any constrained loop non compatibilities until all of the other type of non-compatibilities have been removed (using Algorithm III). If after the other non-compatibilities have be removed, a partition or partitions remain non-compatible, we sum the number of unique arrays referenced for nodes of each color within each remaining non-compatible partition and then, remove from each of these partitions the colored nodes corresponding to the fewest number of references. The effect is to increase the number of arrays required by the number arcs into and out of these nodes.

In Figure 3.8, if Nodes 3 and -4 were each constrained, Node 3 would be removed from the cluster since the fewest number of arrays in the middle partition would be affected, in this case, two arrays, as opposed to three arrays were Node -4 removed Since Node 3 is a source node for this partition, however, it becomes immediately eligible for clustering with Node 1 in the left-most partition. As a consequence, once transformation is completed, the added cost to achieve loop compatibility, in this case, is the cost of just one array.

[5] Recall, a constrained loop is one in which there is a loop-carried dependence

3.2 Analysis of Partitioning Effectiveness

In the previous section we described a method for partitioning a non-compatible cluster of loops into separate fusionable clusters, and in the process we demonstrated the application of this method with several examples illustrating alternative situations which might be encountered. As a final step, we now justify the effectiveness of our loop partitioning method and derive its complexity.

In general, loop partitioning is a difficult problem, and the conjecture is that cluster partitioning too is NP-complete [GOST92]. Nevertheless, the heuristic we described in Algorithm III works well in many instances, as we have shown. Unfortunately, we cannot guarantee this approach produces optimal results in all cases. In the analysis that follows, we therefore present intuitive arguments for why our algorithm frequently succeeds.

We know the flow-augmenting path algorithm for max-flow min-cut produces an optimal partition of a network for a given set of fixed flow capacities. Therefore, if an SG can be effectively transformed into a network with fixed flow capacities assigned in such a way as to accurately reflect the relative cost of loop-cluster partitioning, we are similarly assured the resulting partitioning too is optimal. To establish optimality, we would need to show *1)* our priorities for cutting edges accurately reflect actual partitioning tradeoffs, and *2)* these priorities are respected during application of the flow-augmenting path algorithm. Since we have no evidence to suggest otherwise, we begin under the assumption that the algorithm might, in fact, be optimal. If it is not, we would like to isolate situations in which the algorithm is likely to fail. It is with this motivation that our analysis proceeds.

In the discussion that follows, a *fork node* in the IG is a node with multiple output ports, and likewise a *join node* is a node with multiple input ports. Note that forks in the IG correspond to joins in the SG, and visa versa. We refer to a combination of a fork node, a join node, and the two segments of the graph which connect these nodes as a *cycle*.[6] Cycles in an IG are further classified according to whether or not they are minimal; those consisting of a unique fork and the fewest number of nodes are said to be *fundamental*. Lastly, when we refer to the *segments of a cycle*, it will be to the various chains that connect a fork and a join Note, as a minimum, every cycle contains at least two such segments As further terminology, we distinguish

[6] This definition is slightly different than common terminology in order to take advantage of both the directed nature of the SG, as well as the characteristics of the underlying IG

undirected edges in the IG from directed arcs in the induced SG

By construction, we guarantee that the SG has no cycles of odd length, for we refuse to add arcs to the graph which would produce such cycles Nonetheless, let us assume for a moment we were to add arcs which would introduce odd length cycles Surely we would have to break all of these cycles in order to make the SG two-colorable. At the same time we might need to cut other arcs of the graph to partition it. Partitioning, as we have shown previously, is essential to form fusionable loop clusters. Of course, arcs cannot be cut indiscriminately Consequently, we need a means of establishing priorities with respect to the order in which arcs of the SG can be cut.

As a way of discovering these priorities, let us assume that an IG consists of just a single isolated odd-length cycle. To break this cycle, we must partition it into two sets of nodes. Moreover, we know by inspection that both segments must be broken, and this represents the least cost Here each segment consists of, at most, one chain Obviously, the least-cost cut of each segment, in this case, would be an edge cut But, for the case of an isolated cycle, no edge costs less to cut than the edges extending from the fork. This is because usually only a single array is affected; whereas, any other cut affects at least two arrays Again we emphasize that our ultimate objective is to eliminate as many arrays as possible

In most instances, the array generated within a fork loop is used by multiple adjacent loops; hence, by partitioning the cycle at the fork, two edges are cut at a cost of one array. On the other hand, a partition anywhere else in the cycle implies the generation of at least two arrays, within two loops, each array being passed to its respective consumer loop; hence, two arcs are cut at a cost of two arrays Even when the edges of a fork represent the transfer of different arrays, the fork-partition cost can be no higher than any other combination of cuts Moreover, the preceding least-cost relation holds regardless of how many times a particular array is referenced within a subsequent loop, since the cost of multiple internal array references can always be avoided by making the first use within the loop an assignment to a scalar variable, and the scalar then used to satisfy the remaining references to the array element. Based upon the preceding observations, we conclude that, in the absence of any global criteria, the best location to partition a cycle is across the output edges of a fork

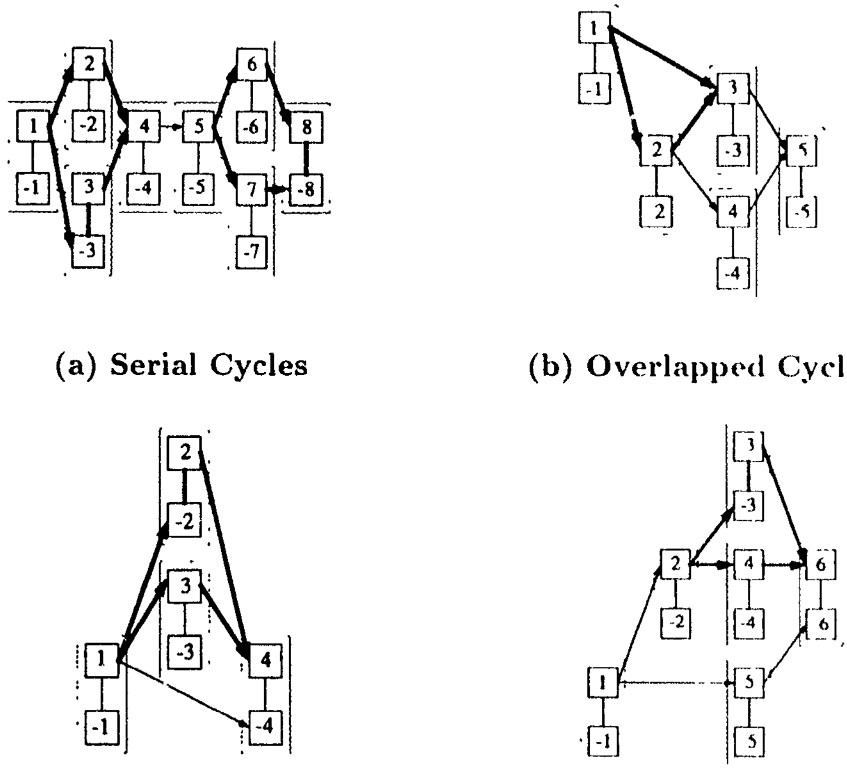
Now let us take a somewhat broader perspective. First we observe that topologically cycles can be viewed as combining in one of several ways serially, in a nested manner, in an overlapped manner, or concurrently.[7] An example of each of these basic structural combinations is illustrated in Figure 3.9. When cycles combine *serially*, the fork node and join node of the first cycle occurs topologically before the fork node and join node of the second, as shown in Figure 3.9(a). In the figure, two odd-length serial cycles appear, each with bold arcs. When cycles are *overlapped*, as shown in Figure 3.9(b), the fork of the first is constrained to occur before the fork of the second, and the join of the first is constrained to occur before the join of the second. In this figure, as in the previous one, an odd-length cycle is indicated by bold arcs. When cycles are *nested*, the fork node and join node of the first cycle are constrained by data dependence to occur between the fork and join of the second cycle, as shown in Figure 3.9(c). Again, the odd length cycle is highlighted in the figure Lastly, when cycles occur *concurrently*, there is a nested constraint on the sequence of forks and joins between the two cycles, as shown in Figure 3.9(d), but the order of internal nodes with respect to an outer cycle is not fixed. Broadly speaking, the cycles in the last three figures all have one thing in common--at least one shared node that is not simultaneously a join in one cycle and a fork in another, and collectively we refer to cycles of this type as *adjacent* cycles.

From examination of Figure 3.9, a few important relationships become clear:

1. Serial cycles require multiple partitions because there is no way to partition a serial combination of odd-length cycles with a single break. Moreover, since serial odd-length cycles involve multiple breaks, such breaks must be accomplished sequentially, either from left to right or from right to left, to reduce the search cost associated with finding other serial cycles, should any remain. As a further consideration, a side effect of partitioning is usually the breaking of additional cycles which otherwise would cause compatibility constraints on the far side of the partition. These last two points will become clear when we later justify the behavior of the labeling portion of the overall algorithm. Because of these reasons, serial cuts must be given highest priority, if both optimality and efficiency are to be achieved.

2. If there exists an outer-level odd-length concurrent cycle which has an inner odd-length cycle within one of its segments, the inner cycle must be broken before the outer cycle since the inner cycle must be cut regardless, cf. Figures 3.9(c) and (d) Moreover, the breaking of an inner cycle always affects the least-cost

[7] These classifications are merely illustrative, since classifying cycles in a graph is like finding images in ink spots



(a) Serial Cycles

(b) Overlapped Cycles

(c) Nested Cycles

(d) Concurrent Cycles

Figure 3.9: Examples of Ways Cycles Combine

partitioning of the outer cycle. Consequently, the cutting of inner-most cycles warrant as much priority as the breaking serial cycles.

3. The cuts of least priority are those that are part of overlapped odd-length cycles, the reason being that cutting such cycles affects fewer cycles in general than other combinations of breaks, cf. Figure 3.9(b). Also, the least-cost break of overlapped cycles is a cut across the arcs of their respective forks, just as it is in the case of a single cycle (as described above). The least-cost cut is never one across a shared arc, as one might suppose. This is because cutting the shared arc still leaves two remaining arcs to be cut Thus the total cost is a minimum of three arrays; whereas, a cut across forks of the respective cycles cuts four arcs—but only affects two arrays.

4. Lastly, in the absence of any of the above global criteria, the least expensive

location to break an odd-length cycle is at its fork, as explained in the preceding paragraph. It must be emphasized, however, breaking a cycle at the fork is a break of "last resort". Even so, it must be accomplished in order to break the odd length cycle if no higher priority cut exists.

We are now ready to see how these partitioning priorities are manifest in the flow capacities assigned to the arcs of the SG. In effect, the priority of cuts in the SG is a function of two factors: *1)* the flow capacity on the respective edges, and *2)* the topological order in which odd length cycles are discovered by the flow-augmenting path algorithm. To see how these factors affect the partitioning, let us reexamine the way odd length cycles are identified and the way in which flow capacities are assigned.

The way the algorithm identifies odd-length cycles is by their respective forks. The fact that the breadth-first traversal of the IG which is used to create the SG always finds the root is not difficult to show: merely perform a breadth-first traversal of any cycle in the graph, beginning from a join node and traversing the graph in reverse direction from the direction of the arcs.[8] If a cycle exists, the separate paths extending from the join node must come together; otherwise, the structure would not be a cycle, and the only possible place these segments can possibly join is at the fork. By not connecting X-arcs to the SG at the fork and by assigning zero capacity to the adjacent arcs, we ensure all odd-length cycles in the graph are broken. This step alone is enough to guarantee partitioning effectiveness, but it isn't enough to ensure optimality. Note, however, that X-cuts always occur at the farthest end of each segment, and the flow-augmenting path algorithm exploits this fact.

The way optimality is achieved, that is, if it is achieved, is by ensuring that a higher priority cut along a segment of an odd-length cycle is always reached by the flow-augmenting path algorithm prior to reaching the lower-priority X-cut at the farthest end of the cycle. As a consequence of this, lower-priority cuts which are "scheduled" to occur later the the cycle, such as as the X-cuts at the fork, are prevented from occurring, whenever a higher-priority cut exists along the same segment.

Flow capacities also have a lot to do with which arcs are cut: *1)* they ensure arcs at the most deeply nested/concurrent odd-length cycle are broken before any outer-level odd length cycles, and *2)* they break arcs at the forks if the cycle has not already been previously broken. To verify this fact, one need only observe that the fork in an outer cycle is always farther away from the source node than any fork nodes of

[8] the edge within each block which connects a primary to a compliment node can be viewed as bi-directional

a corresponding inner cycle. The two characteristics above are exactly the priorities that are necessary for optimality, as we show next.

The reason the flow-augmenting path algorithm is able order the priority of cuts stems from the fact that the SG is an acyclic directed graph which imposes a fixed order upon the way the segments of the graph are traversed in search of augmenting paths. As a result, if a graph admits multiple minimum capacity cutsets, the augmenting-path algorithm always saturates arcs closest to the source first. It is this characteristic of the flow-augmenting path algorithm that the partitioning algorithm exploits, and it is a direct consequence of the way flow capacities are assigned. Cuts which have no meaning in the context of the loops themselves are prevented from occurring by assigning them infinite capacities, such as the arcs which connect the primary node to its respective complement node in a block. Likewise cuts which should never be made along a chain because of other higher priority edges closer to a the fork are also prevented in the same way.

At the same time, the node-capacity labeling methodology ensures no cycle is starved of capacity, unless it is a candidate for cutting. Specifically, it assures sufficient flow capacity reaches the most deeply nested/concurrent cycle, unless flow is prematurely restricted by a prior odd-length cycle. For just a moment, neglect the possibility of serial cycles, and consider only adjacent cycles. The partitioning algorithm always breaks the most deeply nested odd-length cycle at its fork, the least cost location. This break has the added effect of severing one of the two segments in an outer-level cycle, and obviously, the same is true of the second segment of the outer cycle, if an odd-length nested cycle is present along that segment. It too is cut prior to the outer-level fork, thus ensuring the outer-level cycle is cut at its least cost location, at the same time. Whether the outer-level cycle has an X-arc and zero capacity arc at the fork or not, has no effect in this instance because of the existence of higher priority cuts prior to the fork.

Now let us consider serial cycles. As previously emphasized, the flow augmenting path algorithm cuts the graph in a greedy fashion when multiple flows simultaneously reach capacity at several locations at once. Such cuts always occur within the odd-length cycle, or odd-length adjacent cycles closest to the source node of the graph. As a consequence, only nodes on the far side or sink side, of the partition need be searched for additional serial cycles. At the same time, serial cuts honor the same priorities described in the preceding paragraph with regard to internal cycles. Therefore, they too are accomplished in an least cost manner. As a further consequence of serial partitioning, however, the flow-augmenting path algorithm must be run again once for each serial combination of odd-length cycles remaining in the graph. Since doing so is unavoidable, and since each subsequent partitioning is guaranteed to be accomplished in the same least cost manner, the partition algorithm is likely to always find the best place to partition the graph during each succeeding recursion.

As a final step, let us assume we have an arbitrary spanning graph which for which we have established an optimal partitioning, using Algorithm III. If we were to now connect any primary node within this graph to any other node in the graph, the connection would result in one or more new cycles. However, the creation of these new cycles would in no way affect the underlying priority relationships described above. In other words, the above partitioning priorities would remain invariant: Inner most nested/concurrent cycles would always be cut first, and serial cycles would continue to require multiple partitions. Likewise, the capacity weighting scheme and network saturation pattern of the flow-augmenting path algorithm would remain invariant, ensuring that the priority relationships manifested in the new graph too would be respected by the cluster partitioning algorithm. As a consequence, we have strong evidence to suggest that Algorithm III is, in fact, an optimal algorithm. Nonetheless, the relationships involved in graph partitioning are complex (as previously noted), and additional analysis and testing are required to substantiate our conjecture that Algorithm III is optimal - work which might be undertaken as a follow-on to this research.

Lastly, let us examine the complexity of the algorithm. In effect there are two principle parts: graph coloring and graph partitioning. The first, graph coloring, uses a breadth-first traversal of the interference graph to determine loop compatibility, requiring $\mathcal{O}(n + a)$ steps, where n is the number of nodes in the LCG, or loops in the original cluster, and a is the number of arcs in the LCG, or array transfers between loops in the original cluster. The second part, graph partitioning uses the flow-augmenting path algorithm which requires $\mathcal{O}(na)$ steps. Since a coloring and partitioning occur each time the graph is partitioned, the overall complexity of cluster partitioning is $\mathcal{O}(n(na + a))$.

Chapter 4

Related Research

In this chapter we describe alternative methods for determining loop compatibility and for partitioning non-compatible loops into compatible loop clusters. First we examine the collective loop analysis and transformation technique proposed by Sarkar and Gao [SG91]. This technique provides the basis for the technique we described in Chapter 2. The second technique we describe is the collective loop fusion method for non-compatible loop clusters described by Gao, Olsen, Sarkar, and Thekkath [GOST92]. This later technique addresses the partitioning problem examined in Chapter 3. Lastly, we provide a brief review of other related research on the general transformability of loops.

4.1 Loop Analysis and Transformation

The Sarkar-Gao method for collective loop analysis is more general than the method described in Chapter 2 because the target computer architecture can be either a uniprocessor or a multiprocessor, provided the multiprocessor has sufficient processing elements, inter-processor communication channels, and facilities for small communications buffers at the communication ports of each processing element. In the above context, "sufficient" implies the architecture has adequate resources to take advantage of the parallelism exposed by the transformations, therefore, to a compiler the machine is viewed as having unbounded resources. Although unrealistic, this assumption is usually not restrictive since the required parallelism is usually within the capability of modern multiprocessor architectures [Gao90].

For uniprocessors the transformation strategy is to create fusionable loop clusters, as described in Chapters 2–3, but the algorithms described in this chapter differ slightly, and the impact of these differences is examined. As described in Chapter 1, however, software pipelining and other instruction scheduling techniques are then used to exploit the instruction-level parallelism. For multiprocessors the strategy is to situate producer and consumer loops on separate processors with a physical communications buffer between processors to provide a maximal flow of array elements to each processor, without using main memory for intermediate storage. In this case, loop transformation is performed in two steps: *1)* loop nests are transformed to maximize loop compatibility, thereby permitting the pipelining of array elements between loops, and, *2)* buffer requirements are determined which maintain a maximum flow of array elements between producer and consumer loops. For the remainder of this section we focus primarily on the multiprocessor case.

4.1.1 Loop Compatibility Analysis

Sarkar and Gao were the first to formulate the loop compatibility problem as a two-color graph coloring problem, using an interference graph (IG) constructed from a loop communication graph. They also proved that, if a solution exists to the two-coloring problem, it can be used to obtain an optimal loop configuration, and they described a heuristic to find a two-coloring of the reduced graph obtained by selectively removing IG edges. The particular algorithm they proposed for determining loop compatibility is given below.

Algorithm IV *Two-color a Loop Communication Graph (LCG) to minimize the number of non-compatible output ports*

input: An LCG.

output: A loop configuration vector, $(s_1, \ldots, s_k)$, where s_i = "+" if node n_i^+ in the corresponding interference graph (IG) was colored with color c_1 of two colors, and s_i = "−" if node n_i^- in G_I was colored with c_1.

procedure

```
construct an IG from the LCG ;; see Algorithm I, page 15
FOR each edge in the IG
```

```
        IF edge $e_{ij}$ is an $\{n_a^+, n_a^-\}$ edge i.e. if $e_{ij}$ connects nodes within a block[1]
            weight $w_{ij}$ is $+\infty$
        ELSE
            $w_{ij}$ is number of static references within the node at the input port
    bestcost := $+\infty$
    FOR each node $i$ in the IG
        build a maximum-weight spanning tree $T_i$, rooted at node $i$
        color $T_i$ with two colors, starting at a precolored node
        curcost := sum of the weights of all edges connecting the same color nodes
        IF curcost < bestcost
            bestcost is curcost
            store the coloring in colormap
    RETURN colormap to define the output loop configuration vector
```

With the Sarkar-Gao heuristic, edge weights are used to guide the construction of the tree, that is, the edge with the largest weight is always the next edge to be included in the tree. To avoid biases due to the choice of root, the spanning tree algorithm is repeated for all nodes, giving the algorithm a worst case execution time of $\mathcal{O}(|N_I| \times (|N_I| + |E_I|))$. Compared with Algorithm II, the breadth first spanning-tree algorithm described in Chapter 2 (page 18), the Sarkar-Gao technique, using Algorithm IV, is quite costly, recall Algorithm II executes in $\mathcal{O}(N)$ time. The difference in cost is justified in the multiprocessor case, however, since the follow on transformations too are quite different. In the uniprocessor case, the objective is loop fusion and array contraction; therefore, the least-cost arcs to remove in the case of non-compatibility are always those which form the branches of the forks in the LCG. On the other hand, forks have no special significance with respect to the follow on distribution of loops among processing elements in the multiprocessor case, and so the number of alternatives for arc selection is much greater, and the added complexity is useful. Algorithm IV also handles cyclic interference graphs, and hence it is suitable for cyclic LCGs. However, the feedback edges must initially be compatible, otherwise the original program will deadlock. To avoid deadlock, the IG edges corresponding to LCG feedback edges are each assigned a cost of $+\infty$.

[1] The definition of a block within an IG is given in Chapter 2, Section 2.2 (page 13).

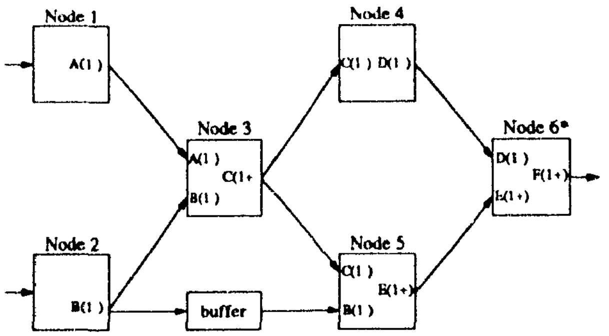


Figure 4.1. Buffer Allocation for an LCG

4.1.2 Buffer Allocation

Buffer optimization is performed only on compatible loops. We illustrate buffering with our original example from Chapter 2, Figure 2.1, page 12. Recall array B is generated by loop 2 and is consumed along two different paths. On the upper path, B is processed first by loop 3, and the result, array C, is sent on to Loop 5. On the lower path, B is routed directly to loop 5. Loop 2 could run to completion before loops 3 and 5 begin, but this would cause the entire array to be stored. On the other hand, the producer (loop 2) and the consumers (loops 3 and 5) are able to execute concurrently [Gao90] As the elements of B are generated by loop 2, they can be immediately consumed by the follow-on loops. Again, this can only occur if the array communication arcs are compatible. If the LCG is cyclic, buffers are used only on the forward arcs and not on feedback arcs.

Since the two paths in Figure 2.1(b) have different execution times, temporary storage is needed along the lower communication path to hold elements of B until they are required by loop 5 In Figure 4.1 we show the LCG after transformation. In this case, a buffer of size w keeps the three loops operating in a maximally pipelined manner, where w is a function of the execution time of upper path

To solve the buffer-allocation problem, it is necessary to assign weights to arcs in the LCG. Let $G = (N, A, W)$ be a weighted LCG with each arc $a = (i, j) \in A$ being weighted by a nonnegative integer $w_a \in W$ When a value produced by node i arrives at an input port of j, it is retained for some number of instruction executions before reaching the output port this number is w_a.

The objective of buffer allocation is to determine requirements for buffers such that the weight of any two distinct paths between any two nodes are equal. An LCG which meets this condition is *balanced*. The algorithm to accomplish the balancing does two things. 1) it locates arcs where buffers are needed, and 2) it determines the size (or weight) of each required buffer. In solving the problem we observe that the heaviest path from node i to j must not change, and that buffers must be inserted only up to the point that all path weights between i and j are the same. As part of the procedure, the weight of the heaviest path from s to i is determined for each $i \in N$. If we let G' be a balanced graph and $w_p()$ and $w'_p()$ be functions which return the heaviest paths for G and G', for any node i, $w_p(i) = w'_p(i)$ must hold. Without loss of generality, we can assume that G has a unique source node s. The first statement in the buffer-allocation algorithm (below) finds the heaviest path weight from the source to all other nodes, using a technique such as Dijkstra's shortest path algorithm [Dij59].

Algorithm V *determine requirements for external buffers*

input: A weighted LCG, $G = (N, A, W)$ having a unique source node s

output: A balanced graph $G' = (N, A, W')$

procedure:

```
compute the heaviest-path $w_p()$ for $G$
FOR each arc $a = (i,j) \in A$
    insert a buffer of size $w_p(j) - w_p(i) - w_a$ along arc $a$
    add this size to $w_a$ to obtain $w'_a = w_p(j) - w_p(i)$, the new weight in $W'$
RETURN $G'$
```

The overall complexity of the shortest-path algorithm is $\mathcal{O}(|N||A|)$, but since node degrees are usually small, $|A| \simeq |N|$. Therefore, for practical purposes the algorithm has a complexity of $\mathcal{O}(|N|^2)$.

4.2 Collective Loop Fusion

Collective loop fusion is a loop partitioning technique which attempts to find a set of loop clusters which minimizes the number of inter-cluster arcs. It is an alternative technique to the partitioning technique described in Chapter 3. Although the technique is oriented toward uniprocessor applications, the same performance benefits accrue for multiprocessor applications as well, especially if limited resources will not permit complete distribution of the original loops among processors. The objective of collective loop fusion implicitly assumes a cost savings from each compatible-contractable arc. For example, assume this savings is 1 unit, denoting the cost of an array-element load from memory. The storing cost is ignored because *1)* whether it can be eliminated depends upon whether the element is used later in the program and *2)* data storage is buffered in most modern uniprocessor architectures. It is also possible for a store or a load to be contained within a conditional, in which case the arc cost would be less than 1, based upon the branching probabilities of the respective conditionals [Sar90b].

The program representation used in the following algorithm is a *Loop Dependence Graph* (LDG). An LDG is essentially an LCG, with the following differences: The arcs in an LDG correspond to *flow*, *anti*, or *output* data dependences [Wol89], and only flow arcs can be compatible. In an LDG, nodes $n_1, \ldots, n_k$ are numbered according to the topological execution ordering of the loop nests in the original loop collection. Also, since the loop nests $n_1, \ldots, n_k$ are assumed to be identically control dependent, there can be no control branching among these loops.

As a means of illustration we will use the collection of FORTRAN loops shown in Figure 4.2. The LDG for this collection is shown in Figure 4.3(a). Non-compatible arcs are marked in figure with an X. The non-compatible arc from n_1 to n_3 is due to array A being produced in one order and consumed in the reverse order, and similarly, the non-compatible arc from n_4 to n_6 occurs due to array E being produced and consumed in reverse orders. The non-contractable arc is marked with an asterisk. A non-contractable arc is one representing an array which cannot be contracted even when loop fusion is possible. The non-contractable arc from n_1 to n_4 is due to an anti dependence on array E. All other arcs in the LDG are both compatible and contractable. Collective loop analysis, as described earlier in this chapter or as described in Chapter 2, is used to determine arc compatibility, and data dependence analysis, as described by Wolfe, is used to reveal array contractability [Wol89].

```
      DO 10 I=1, N
10       A(I) = E(I)
      DO 20 I=1, N
         B(I) = A(I) * 2 + 3
20       C(I) = B(I) + 99
      DO 30 I=1, N
30       D(I) = A(N-I+1) + 6
      DO 40 I=1, N
40       E(I) = B(I) + C(I) * D(I)
      DO 50 I=1, N
         F(I) = B(I) * 4 + 2
50       G(I) = E(I) * 8 - F(I)
      DO 60 I=1, N
60       H(I) = F(I) + G(I) * E(N-I+1)
```

Figure 4.2: Sample Collection of Non-Compatible FORTRAN Loops

Collective loop fusion works on any acyclic graph having an unrestricted number of non-compatible and/or noncontractable arcs. Although the general collective loop fusion proolem is probably NP-complete, the algorithm we describe is optimal for bi-partitioning - a case for which the *flow-augmenting path* algorithm yields a cutset of minimum size [GOST92]. The objective of collective loop fusion is to find the minimum number of clusters in which all non-compatible arcs are separated across clusters. According to Gao, Olsen, Sarkar, and Thekkath, this will maximize the number of contractable edges within each cluster [GOST92]. The overall strategy consists of two phases: *1)* determining a set of clusters to which each node could feasibly belong, and *2)* selecting nodes from these sets and assigning them to actual clusters. This later phase is then repeated until all nodes have been assigned.

The first phase of the algorithm is accomplished by scanning nodes first in a forward topological sequence to determine the earliest cluster to which each node could belong and then scanning the nodes in a reverse topological sequence to determine the remaining clusters to which each could belong. The second phase of the algorithm is also a two-step process: First, nodes which could only belong to a single cluster are removed from the sets of feasible clusters and assigned immediately to an actual cluster. This leaves the nodes which could belong to more than one cluster. For these nodes, a least-cost determination is made as to which actual cluster each node should be assigned. This later determination is made in a iterative manner using the flow-augmenting path algorithm, which is based upon the max-flow min-cut theorem (described in Chapter 3).

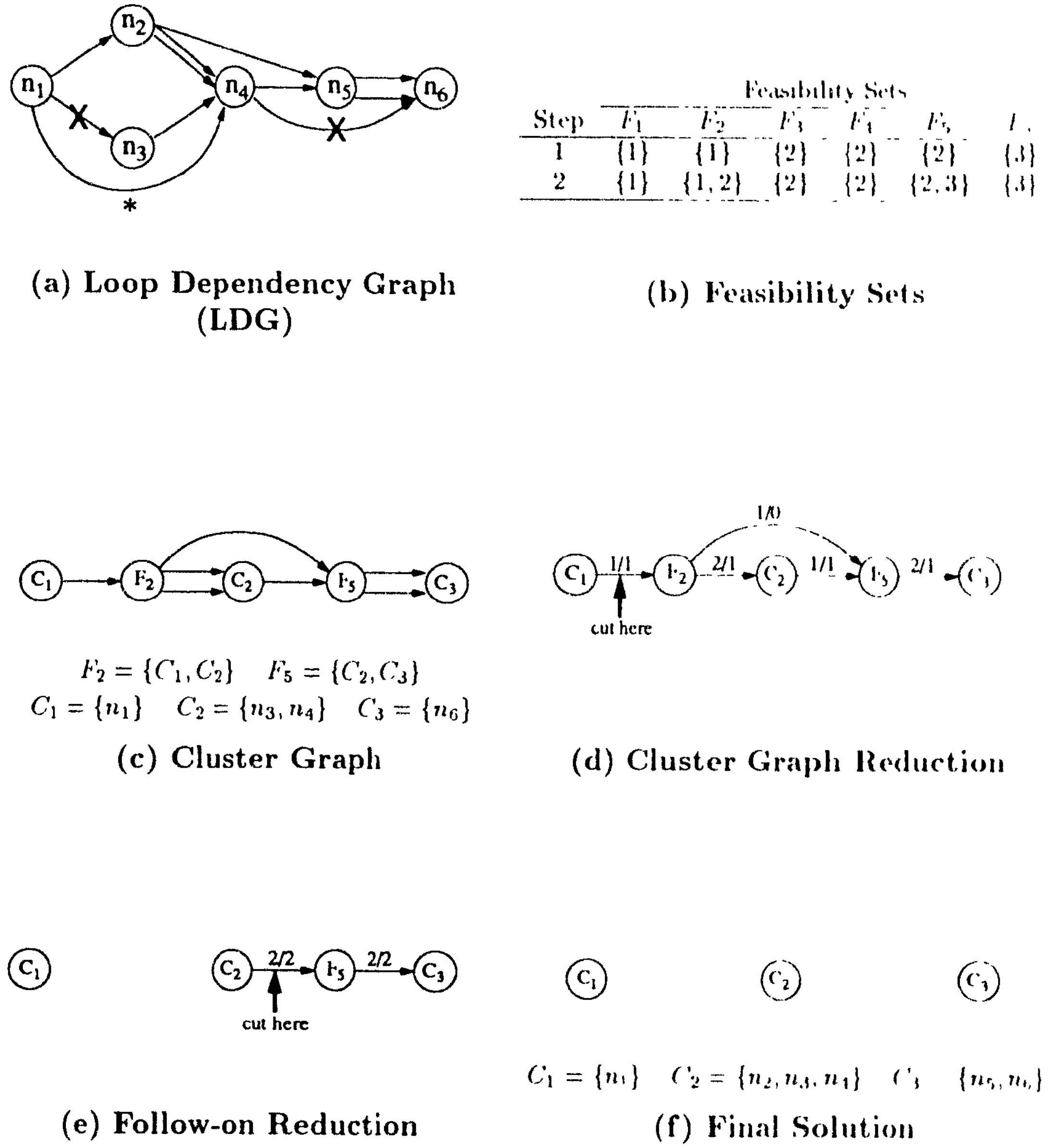
4.2.1 Terminology and Methodology

Before formally presenting the algorithm for collective fusion, we need to explain some additional terminology and give a general description of the overall methodology.[2] The input to the algorithm is an LDG, say $G = [N, A]$, in which all nodes $n \in N$ have been assigned unique labels according to their topological order. As indicated above, the first step is to determine the *feasible clusters* to which each node might belong. This information forms a set $F_n = \{c \mid 1 \leq c \leq k\}$ for each node, where n is the topological number of the node, c is the number of a cluster to which n could belong, and k is the minimum number of clusters that are necessary to partition the non-compatible loop collection represented by G. To determine the members of these *feasibility sets*, the algorithm first visits each node in topological sequence and determines the lowest-numbered cluster to which each node can belong. For source nodes in G, $F_{src} = \{1\}$, indicating the earliest cluster to which each can be assigned is cluster C_1. If the input arc of a node is a non-compatible arc (marked with an X), the earliest cluster is one more than the cluster number at the origin of that arc. Likewise, if a node n is a terminus to multiple arcs, the highest cluster number of any origin incident an input arc of n determines F_n. If the maximum cluster number assigned to any feasibility set is k, then k is the minimum number of clusters required to partition the graph. In the table in Figure 4.3(b) we show for Step 1 the feasibility sets for each node following forward traversal of the LDG in Figure 4.3(a). In this case, G requires three clusters, C_1, C_2, and C_3.

As a follow-on step, the nodes of G are traversed in reverse topological sequence to determine the last and any possible intervening clusters to which each node could belong. These cluster numbers too are added to the feasibility sets of the respective nodes. As a result of this step, the feasibility set for any particular node can contain several possible clusters. Multiple node feasibility sets can be seen, for example, in Step 2 of the table in Figure 4.3(b); they are sets F_2 and F_5.

Feasibility sets are used to create a reduced graph $G' = [N', A']$, called a *Cluster Graph* (CG). G' is constructed from G by consolidating nodes and arcs in G. In Figure

[2] Our terminology and methodology are somewhat different from that described by Gao, Olsen, Sarkar, and Thekkath [GOST92]



(a) Loop Dependency Graph (LDG)

Feasibility Sets

Step	F_1	F_2	F_3	F_4	F_5	F_6
1	{1}	{1}	{2}	{2}	{2}	{3}
2	{1}	{1, 2}	{2}	{2}	{2, 3}	{3}

(b) Feasibility Sets

$F_2 = \{C_1, C_2\}$ $F_5 = \{C_2, C_3\}$

$C_1 = \{n_1\}$ $C_2 = \{n_3, n_4\}$ $C_3 = \{n_6\}$

(c) Cluster Graph

(d) Cluster Graph Reduction

(e) Follow-on Reduction

$C_1 = \{n_1\}$ $C_2 = \{n_2, n_3, n_4\}$ $C_3 = \{n_5, n_6\}$

(f) Final Solution

Figure 4.3. An Example of Cluster Partitioning

4.3(c) we show the CG for the example in Figure 4.3(a). The nodes in a CG represent two distinct situations *1)* nodes that have already been been assigned to clusters and *2)* nodes awaiting to be assigned. Each of these categories is represented in the CG by a different type of node. The first type is initially formed from singleton feasibility sets. These sets contain only a single cluster indicating there was no alternative cluster to which the node could be assigned; we indicate sets of this type by $F_n^1 = \{F_n \mid |F_n| = 1\}$. The complementary set of feasibility sets are those containing nodes that could belong to more than one cluster; these sets are indicated by $F_n^{>1} = F - F_n^1$, where $F = \{F_n \mid 1 \leq n \leq |N|\}$. All of the consolidated nodes $n' \in N'$ in G' are created from either one or the other of these two types of sets. The first type of node in N' actually represents a cluster of nodes from G, indicated by $C_i = \{n \mid n \in F_n^1\}$. For example, in Figure 4.3(c), cluster node $C_2 = \{n_3, n_4\}$ was created from feasibility sets F_3 and F_4, cf., Figure 4.3(b). The second type of node in G' is a node F_n, formed from the corresponding feasibility set $F_n^{>1}$ for n. In Figure 4.3(c), for example, feasibility node F_2 corresponds to feasibility set F_2 in Figure 4.3(b), and likewise node F_5 corresponds to set F_5. The set of arcs in G' is $A' = \{(m,n) \in N' \times N' \text{ s.t, } \{m,n\} \not\subseteq C_i \text{ and } (m,n) \text{ is contractable}\}$. The arcs not included in N' therefore are *1)* those that are eliminated because they would be internal to one of the newly formed cluster nodes and *2)* those that are eliminated because their edge-capacity is zero, such as non fusible arcs (marked by an asterisk).[3]

Overall, the construction of the clester graph achieves two purposes. *1)* it consolidates nodes in the LDG that must belong to the same cluster, and *2)* it eliminates superfluous arcs that are not required in making future partitioning decisions. The CG is also the input for the Phase 2 transformation, which we describe next. Before we describe Phase 2, however, we formally state Phase 1:

Algorithm VI (Phase 1) *Create a Cluster Graph from a Loop Dependence Graph (LDG).*

Input An acylic graph LDG $G = [N, A]$.

Output: An acyclic cluster graph $G' = [N', A']$.

[3] Recall that non-fusible arcs are also non-contractable

Procedure:

```
;; traverse the nodes of G in topological order
    FORALL $n \in N$
        IF $n$ has no incoming arcs
            $F_n = \{1\}$
        ELSE
            FORALL input arcs $(m,n)$ incident to $n$
                $F_n = \mathtt{MAX}\left(\begin{cases} \{i+1\},\ C_i \in F_m, & \text{if } (m,n) \text{ is non compatible} \\ F_m, & \text{otherwise} \end{cases}\right)$
;; traverse the nodes of the graph in reverse topological order
    FORALL $m \in N$
        IF $\exists(m,n)$ s.t. $(m,n)$ is non-compatible
            $x_m = \mathtt{MAX}(i \in F_m)$
            FORALL output arcs $(m,n)$ incident to $m$
                $x_n = \mathtt{MAX}(j \in F_n)$
            IF $x_m < \mathtt{MIN}(x_n)$
                FORALL $j$ s.t $x_m < j \leq x_n$
                    add $j$ to $F_m$
;; construct G', having N' and A' defined as follows:
    FORALL $F_n \in F$
        IF $i \in F_n^1$
            $C_i \in N'$, where $C_i = \{n \mid i \in F_n\}$
        ELSIF $F_n^{>1}$
            add $F_n$ to $N'$
    FORALL $n' \in N'$
        IF $\{m,n\} \not\subseteq n'$ AND $(m,n)$ is contractable
            add $(m,n)$ to $A'$
```

The CG initially consists of a single component which is iteratively reduced as compatible clusters are identified and removed Feasibility nodes in the graph are also collapsed into adjacent cluster nodes, as part of the cluster identification process. To collapse a feasibility node, the node itself is removed and the node in the LDG which it represents, i.e., the node indicated by the F-subscript, is added to the adjoining cluster to which it is being assigned To ensure cluster assignment is accomplished in a cost effective manner. Algorithm VI uses the flow-augmenting path algorithm to determine to which adjacent cluster node the LDG node referenced by the feasibility node should be assigned. The collective-loop-fusion problem is transformed into a network flow problem by assigning a capacity of 1 to each arc in the CG. Multiple arcs between adjacent nodes in the CG are also collapsed where possible and arc capacity adjusted. The flow augmenting path algorithm returns a minimum partition containing the fewest number of arcs. The flow-augmenting path algorithm is repeatedly applied on the primary component until the CG is reduced to only a single node.

When the partition isolates a feasibility node, as opposed to a cluster node, the feasibility node becomes a new component in the the reduced CG which continues to be considered each time a new cluster node is removed, possibly removing a cluster reference from the isolated feasibility node. At the stage the feasibility node contains only one remaining cluster reference, it is eliminated, just as other feasibility nodes within the primary component, and the corresponding node referenced in the LDG is assigned to the remaining referenced cluster, just as in the preceding case.

Each application of the flow-augmenting path algorithm partitions the graph into two components, each smaller than the original. Sometimes the CG can be partitioned simultaneously into more than two components by cutting the graph at every arc for which the flow capacity has been reached, not just the minimum cutset. Two cases in which the graph cannot be cut in this way are: *1)* when a cluster node is disconnected from the other nodes, and *2)* when removal of an arc would not disconnect the graph. The following formally describes Phase 2 of the algorithm.

Algorithm VI (Phase 2) *Partition a Cluster Graph (CG) into compatible clusters*

Input The reduced graph $G' = [V', A']$.

Output: A set of compatible clusters.

Procedure:

```
    FORALL $n' \in N'$
    .. collapse and label arcs in $G'$
```

```
        IF multiple compatible arcs $(m, n)$ are incident to $n$
            delete all but one of these arcs
            flow capacity $c_{mn} = 1 +$ number of $(m, n)$ removed
    WHILE $|N'| > 1$ ;; $G'$ is connected
    ;; remove $C_i$ from $G'$
        execute the flow-augmenting path algorithm to determine a partition
        remove the partition from $G'$
        IF the partition is $\{C_i, F_n\}$
            add $n$ to $C_i$
        ELSE
            add $n$ to $C_j$ where $j \in F_n$
        add $C_i$ to the solution
        FORALL $F_n \in G'$
            remove $C_i$
    RETURN the solution
```

In Figures 4.3(d)–(f) we show the application of Phase 2 of Algorithm VI to the CG in Figure 4.3(c). The capacity and flow following application of the flow augmenting path algorithm is shown for each arc in Figure 4.3(d); for each arc, capacity is shown to the left of the slash and flow is shown to the right. Observe that the two arcs between F_2 and C_2 were replaced by a single arc having a capacity of 2. A similar replacement was made between nodes F_5 and C_3. The maximum flow initially occurred between C_1 and F_2, allowing n_2 to be added to C_2 (see Figure 4.3(d)). The arc between C_2 and F_5 creates another arc between C_3 and F_5, and this is replaced by a single arc of capacity 2. In Figure 4.3(e) we show the application of the flow augmenting path algorithm to the remaining graph. This final step places n_5 into cluster C_3, terminating the algorithm.

Certain types of LDGs require special treatment in order to be handled by Algorithm VI. For example, the LDG might contain a set of non fusible nodes, such as n_1 and n_4 in Figure 4.3, which each must be put into separate clusters. These nodes are used during the assignment of cluster numbers, but they are not essential to the application of the flow augmenting path algorithm. Therefore nodes of this type are deleted from all cluster nodes in the CG, as are the arcs to and from these nodes. This isolates these nodes so that the flow-augmenting path algorithm can be correctly applied. For situations in which the LDG is disconnected, Algorithm VI is applied to each component, and once all components are partitioned, the clusters having the same cluster number are merged. Another situation which requires special treatment is one in which there are multiple sources or multiple sinks. In these cases, a *pseudo source* is created having infinite-capacity arcs to all real sources. Similarly, a *pseudo sink* is created having infinite capacity arcs from all real sinks to the pseudo sink.

4.2.2 Algorithm Complexity

The best way to determine the complexity of Algorithm VI is to examine the complexity of each phase of the algorithm separately. First, let us assume that the input LDG has n nodes and a arcs. Phase 1 is composed of three steps. The first is the forward pass of the nodes which also examines incoming arcs. Since each node and each arc are visited once $\mathcal{O}(n + a)$ steps are required. The second step is the reverse pass of the graph which visits each node and the arcs leading out of these nodes. In the worst case all nodes are traversed and all arcs are examined, requiring $\mathcal{O}(na)$ steps. In the third step the CG is built, requiring a visit to all nodes (to build the node set) and a visit to all arcs (to build the arc set), requiring $\mathcal{O}(n + a)$ steps. The total time is $\mathcal{O}(n + a) + \mathcal{O}(na) + \mathcal{O}(n + a) = \mathcal{O}(na)$. To find the complexity of Phase 2, we assume that in the worst case each application of the flow-augmenting path algorithm isolates only a single node, requiring $n - 1$ applications. If the algorithm takes $\mathcal{O}(na \log n)$ [Tar83] the total time is $\mathcal{O}(n^2 a \log n)$. Therefore, the total time for the entire Algorithm VI is $\mathcal{O}(na) + \mathcal{O}(n^2 a \log n) = \mathcal{O}(n^2 a \log n)$.

4.2.3 A Case of Non-Optimal Partitioning

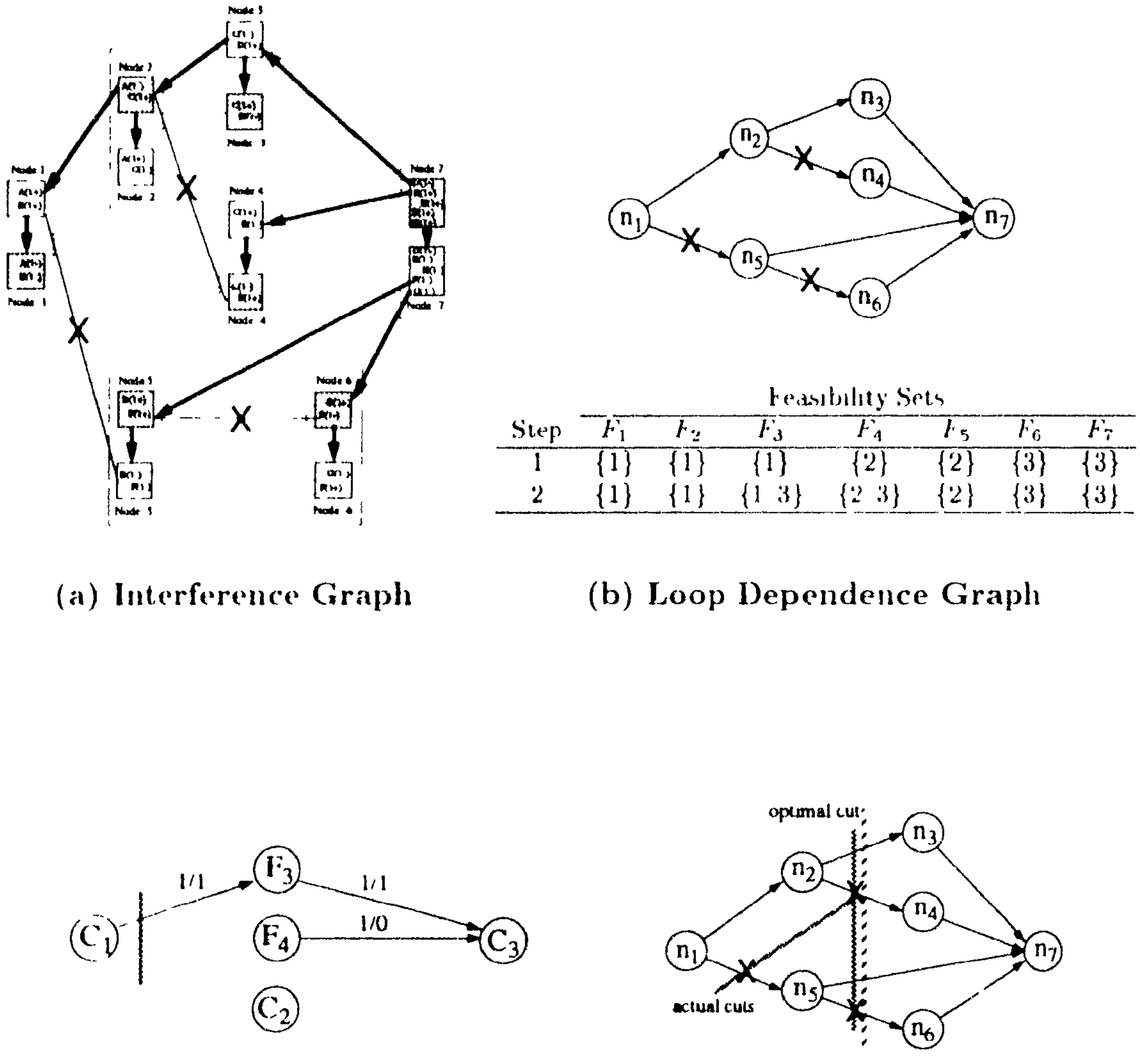
As explained at the beginning of this section, determining an optimal partitioning is likely to be yet another NP-Complete problem, and although the heuristic algorithm we have just described produces an optimal partitioning in many instances, there are instances which are equally likely to occur for which the algorithm produces non-optimal results. In this section we give one such example. The code for this example is shown in Figure 4.4. The example loop collection in this case is comprises seven loops and uses eight arrays. The LDG for this code is shown in Figure 4.5(a). Observe that the collection has three non-compatible arcs, (n_1, n_5), (n_2, n_4), and (n_5, n_6), identified by earlier loop analysis to be non compatible (as described in Chapter 2 or in the first part of this chapter).

```
          DO 10 I=1, N
             A(I) = 36
 10          B(I) = A(I)
          DO 20 I=1, N
 20          C(I) = A(N-I+1) + 99
          DO 30 I=1, N
 30          D(I) = C(N-I+1) + 6
          DO 40 I=1, N
 40          E(N-I+1) = C(I) / 4
          DO 50 I=1, N
 50          F(I) = B(I) * 8 + 2
          DO 60 I=1, N
 60          G(I) = E(N-I+1) - N
          DO 70 I=1, N
 70          H(I) = D(N-I+1) * F(I) + F(I) * G(I)
```

Figure 4.4: Another Collection of Non Compatible FORTRAN Loops

The forward and reverse traversals of the LDG, required in Phase 1 of the algorithm, produces the feasibility sets shown in Figure 4.5(b). For this particular collection of loops, the algorithm identifies an overall requirement for three clusters. However, this overstates the requirement since only two clusters are actually required for optimal partitioning, as we will show later.

The CG which is formed from the feasibility sets (identified in the table) and from the induced set of arcs from the LDG is shown in Figure 4.5(c). In this particular CG there are two feasibility nodes, F_3 and F_4, and three cluster nodes C_1-C_3, created from the fusibility sets, and there are three induced edges taken from the LCG. The arc capacities required to make the graph conformable for network flow analysis are also shown in Figure 4.5(c), in addition to the cutset resulting after application of the flow-augmenting path algorithm. In this instance, only one iteration of the flow-augmenting path algorithm is needed to partition the graph, and network saturation is accomplished with only one path augmentation. Feasibility node F_4 is immediately collapsed so that n_4 can be added to C_3, since this is the only cluster to which F_4 is connected. Feasibility node F_3, however, could go in either C_1 or C_2. The decision in this case is determined by application of the flow augmenting path algorithm, which results in a cut between C_1 and F_3 as shown in Figure 4.3(c). With this cut



(a) Interference Graph

Step	Feasibility Sets						
	F_1	F_2	F_3	F_4	F_5	F_6	F_7
1	$\{1\}$	$\{1\}$	$\{1\}$	$\{2\}$	$\{2\}$	$\{3\}$	$\{3\}$
2	$\{1\}$	$\{1\}$	$\{1\ 3\}$	$\{2\ 3\}$	$\{2\}$	$\{3\}$	$\{3\}$

(b) Loop Dependence Graph

$F_3 = \{C_1, C_2, C_3\}$ $F_4 = \{C_2, C_3\}$
$C_1 = \{n_1, n_2\}$ $C_2 = \{n_5\}$ $C_3 = \{n_6, n_7\}$

(c) Cluster Graph

(d) Optimal and Non-Optimal Partitions

Figure 4.5 A Case of Non-Optimal Partitioning

```
          DO 10 I=1, N
             a = 36
             B(I) = a
10        C(N-I+1) = a + 99
          DO 20 I=1, N
20           F(I) = B(I) * 8 + 2
          DO 30 I=1, N
             d = C(I) + 6
             e = C(N-I+1) / 4
             g = F(N-I+1)  N
30           H(I) = d * e + F(I) * g
```

(a) Non-Optimal Solution

```
          DO 10 I = 1, N
             a = 36
             b = a
             C(N-I+1) = a + 99
10           F(I) = b * 8 + 2
          DO 20 I = 1, N
             d = C(I) + 6
             e = C(N-I+1) / 4
             g = F(N-I+1)  N
20           H(I) = d * e + F(I) * g
```

(b) Optimal Solution

Figure 4.6 Non Optimal and Optimal Partitioning Solutions

F_3 can be collapsed and n_3 added to C_3. The result, therefore, is the partitioning $C_1 = \{n_1, n_2\}$, $C_2 = \{n_5\}$, and $C_3 = \{n_3, n_4, n_6, n_7\}$, as shown in Figure 4.5(d).

Observe that the partitioning produced in this instance is non optimal. As shown in Figure 4.6(a), three loops were needed and four of the original seven arrays were eliminated, whereas, in the optimal solution, shown in Figure 4.6(b), only two loops are actually required, and five of the original arrays could actually be eliminated.[4] The reason for the non-optimality of the collective loop fusion algorithm is explained by the fact that the algorithm implicitly assumes that non compatibility of a non compatible X edge is determined entirely by the relationship between the end nodes of the particular edge and is unaffected by other cuts made during the partitioning process, and therefore, fixed prior to partitioning, whereas, in fact, this is not actually the case, as explained in Chapter 3.

4.3 Other Related Techniques

Most previous work in optimizing the performance of loops has focused on individual loops rather than on collections of loops [KRP+81, AK87, Wol89]. Only recently has

[4] Incidentally, Algorithm III, described in Chapter 3 (page 45), is able to find the optimal partition in this case.

a combination of loop optimization techniques been applied together and only in the case of a single loop nest, specifically, unimodular transformations [Ban90, WL90].

Loop distribution and loop fusion are well known loop transformations to handle multiple loop nests [Wol89]. Loop distribution is typically performed maximally (II-block partitioning), and loop fusion is used as follow up to selectively merge the distributed loops back together. In general, loop fusion reduces loop overhead in uniprocessor codes increasing the size of loop bodies. This affects instruction-scheduling effectiveness and reduces memory requirements, in turn, reducing memory-register data traffic. This last benefit is especially important for vector machines for which vector loads and stores are costly. In spite of these benefits, the ability to effectively perform loop fusion has remained a difficult problem [All83].

Wolfe used the term "array contraction" for a different optimization than the one described here. His use of the term was to describe the function of reducing the size of a compiler generated temporary array created by scalar expansion [Wol89]. Allen also investigated the problem of minimizing temporary array storage in a single loop nest, using a technique called *sectioning*, however, the technique was not extended for optimization beyond a single loop nest [All83]. Callahan et al., have studied register allocation of subscripted variables (arrays), but as with the other related research, their work too focused on single loop nests, as opposed to collections of loops [CCK90].

The opportunity for optimizing array operations across a collection of loops for *dataflow software pipelining* was recognized in Gao [Gao86, Gao90], and loop parallelization and loop chunking across collections of loops has been described by Sarkar [Sar89, SG90, Sar90a]. In our work and in recent work by Gao, Olsen, Sarkar, and Thekkath, collective loop optimization was considered both at the individual loop-nest level (e.g., loop reversal and loop interchange) and at a global level, where loops are partitioned into clusters for fusion [GOST92].

Chapter 5

Relevant Code Transformations

In this chapter we describe the transformations required for analysis of loop clusters, along with code-improving transformations that might be applied once a cluster, or clusters, of transformable loops has been found. Although most of the techniques we describe are generally well known, they are nevertheless important within the current context, and our immediate aim is to demonstrate their relevance.[1] We begin the chapter with description of the transformation algorithms used prior to and during cluster analysis. Of particular importance are loop normalization, direction reversal, and loop interchange.[2] In addition to these transformations, we discuss other transformations which might also be necessary to improve the effectiveness of cluster analysis, two such examples are scalar forward substitution and scalar renaming. Once the transformations required for cluster analysis have been described, we continue by surveying several transformation algorithms that are then possible to achieve improved code performance. The particular transformations we discuss are loop fusion, array contraction, and software pipelining. Lastly, we conclude with a discussion of a few incidental transformations, such as loop unrolling and instruction scheduling, which could, in certain instances, affect loop analysis and/or the effectiveness of any consequent code-improving transformation that is applied.

[1] Detailed information regarding the more common transformations can be found in any one of several sources [ASU86, PW86, Pol88, Wol89, ZC90], however, when appropriate, we cite the principle references following the respective transformation description.

[2] Transformations, such as these, which are of particular importance to loop cluster analysis are appropriately indicated in the text.

5.1 In-Line Expansion

The first transformation we describe is in-line expansion. In-line expansion is important for collective analysis because it exposes loops to analysis which otherwise might not be considered. Expansion provides other benefits too, such as reducing program execution time by eliminating procedure calls (and their associated parameter transfer) and procedure returns. Below we show two different situations in which in-line expansion might be beneficially applied.[3]

Calls within a Loop

```
      DO 10 I = 1, N
10       CALL ADD (A(I),B(N-I+1),C(I))
      END

      SUBROUTINE ADD(X,Y,Z)
         REAL X, Y, Z
         X = Y + Z
      END
```

A Series of Calls

```
      CALL ADD (A,B,C)
      CALL ADD (C,A,D)
      END

      SUBROUTINE ADD(X,Y,Z)
      REAL X(N), Y(N), Z(N)
      DO 10 I = 1, N
10       Z(I) = X(I) + Y(I)
      END
```

Following in-line expansion, the respective code segments become

```
      DO 10 I = 1, N
10       C(I) = A(I) + B(N-I+1)
```

```
      DO 10 I = 1, N
10       C(I) = A(I) + B(I)
      DO 20 I = 1, N
20       D(I) = C(I) + A(I)
```

5.2 Normalization

We can separate normalization transformations into basically two categories: loop normalization and subscript normalization. Although it is the first of these categories that we are primarily concerned with during loop cluster analysis (and to which we devote the most attention), the second category, subscript normalization, is also important, since it too affects the eligibility of loops to undergo further analysis in certain instances. We begin with loop normalization.

[3] In these examples and all subsequent examples we assume **N** is a globally defined loop upper bound, unless otherwise stated.

5.2.1 Loop Normalization

The purpose of loop normalization is to transform a loop into a form in which the loop lower bound and increment are each 1. Normalization is performed early in the loop selection process since it facilitates subsequent loop analysis such as bounds checking. To add concreteness to the technique, but without loss of generality, we assume a typical FORTRAN syntax for loop constructs. With this assumption, the algorithm, which essentially performs index substitution, is shown below [ZC90].

Transformation 5.1 (Loop Normalization) *Transform a loop so that the lower loop bound and loop increment are both 1*

input A FORTRAN loop, within an eligible loop cluster, which has a lower loop bound and/or a step other than 1. The loop is assumed to be of the form

DO *label* I = e_1, e_2, e_3

label *body*

where I is an integer and e_j, for $1 \le j \le 3$, each an expression. e_1 is the lower bound, e_2, the upper bound, and e_3, the loop increment, or step. Variables, t_j, for $1 \le j \le 3$, are temporaries of integer type

output: A normalized loop of the form

$t_1 = e_1$

$t_2 = e_2$

$t_3 = e_3$

DO *label* I′ = 1, $(t_2 - t_1 + t_3)/t_3$, 1

label. *body′*

I = t_1 + MAX($(t_2 - t_1 + t_3)/t_3$,0) * t_3

where *body′* is *body* with $t_1 + (\mathrm{I}' - 1) * t_3$ substituted ∀I

Note that the prelude statements to the loop can be eliminated when all e_j are integer variables or constants. In the case of constants, the constant value is propagated. Likewise, when the sign of a normalized upper bound is known in advance, `MAX` can be replaced by the appropriate text, either the expression or zero, whichever is greater. Lastly, note that the final statement restores I to its original value, a condition that often might not be necessary. When the statement is not needed, it is eliminated by dead-code elimination [ASU86]. The following is a typical example of a loop requiring normalization.

```
      N = 100
      DO 10 I = N-1, 0, -1
10       A(I) = I
```

The normalized version of this loop is shown next, both before and after simplification In this example we assume the loop index variable **I** is dead upon loop exit.

After Normalization

```
      N = 100
      T1 = N - 1
      DO 10 I = 1, (0-T1-1)/-1, 1
10       A(T1+(I-1)* 1) = T1 + (I - 1) * -1
      I = T1 + MAX((0-T1 1)/-1, 0) * -1
```

After Simplification

```
      DO 10 I = 1, N, 1
10       A(N-I) = N - I
```

5.2.2 Subscript Normalization

Subscript normalization usually consists of the following transformations. scalar forward substitution, induction variable substitution, wrap-around variable substitution, and standardization of subscript expressions To understand their relevance to loop cluster analysis, we now describe each transformation in order

Scalar Forward Substitution

In is a natural tendency for programmer to reduce the complexity of expressions, as well as redundant use of subexpressions by separating indexing expressions from the subscript expression in which these expressions are used. For example, a programmer might choose to write

```
      DO 10 I = 1, 100
         J = N - I + 1
10       A(I) = B(J)
```

Unfortunately, statements of this type introduce a forward dependency within the loop which prevents direction reversal. To correct this situation, the compiler substitutes the expression into each index expression within the current scope of the indexing definition

Induction Variable Substitution

For the same reason a programmer might separate lengthy indexing expressions from the array references in which these expressions are used, a programmer might also use a separately computed induction variable Two particular cases are *1)* direction reversal and *2)* loop incrementing A likely case when a programmer might use direction reversal is one involving, say, a copy operation

```
          J = N + 1
          DO 10 I = 1, N
             J = J - 1
10           A(I) = B(J)
```

As before, the solution to this problem is obtained through substitution Specifically, the compiler eliminates the statement that defines the induction variable and replaces each use of the variable with an expression of the form $C_{induction} * I + J_{init}$, where $C_{induction}$ is the induction constant, I is the loop index variable, and J_{init} is the expression for the induction variable initialization When this transformation is applied, the transformed code becomes

```
          DO 10 I = 1, N
10           A(I) = B(N I+1)
```

The second case, loop incrementing, might be used in lieu of using the loop control statement to step through an array

```
          J = 0
          DO 10 I = 1, N
             J = J + 2
10           A(J) = 1 0
```

Although less common today, constructs of this type were once typical due to older FORTRAN language specifications The solution to this particular problem is again the same, and the transformed code in this instance becomes

```
          DO 10 I = 1, N
10           A(2*I) = 1 0
```

Note that the consequence of having a separate induction variable computation is the same as the case involving scalar forward substitution cluster analysis and transformation is inhibited unless the the troublesome statement can be removed, eliminating the original data dependencies

Wrap-Around Variable Substitution

Another common type of situation which results in a restrictive data dependence is the case of wrap-around variables within the loop. A wrap around variable is one that is defined near the end of a loop and is used during the subsequent loop iteration, thus creating a loop carried dependence The following loop provides such an example·

```
         J = N
         DO 10 I = 1, N
            A(I) = B(J)
10          J = J - 1
```

Neither scalar forward substitution nor induction variable substitution alone solves the problem in this instance. But, by rolling the loop back one iteration (adjusting J_{init} and moving the induction statement to the head of the loop) and then performing induction variable'substitution, the loop can be fixed (as shown below) so that collective transformation can proceed unconstrained.

After Loop Rolling

```
         J = N + 1
         DO 10 I = 1, N
            J = J - 1
10          A(I) = B(J)
```

After Variable Substitution

```
         DO 10 I = 1, N
10          A(I) = B(N-I+1)
```

5.3 Scalar Renaming

Scalar renaming is a technique for removing anti dependencies which otherwise prevent certain loops from being considered for collective analysis. The benefit of scalar renaming can be seen in this next example. First, let us assume we have the following loop cluster.

```
a = 1.0
A(1:N:1) = a
a = 5.0
B(1:N:1) = A(N-1:1) + a
```

Notice that redefinition of the scalar variable a, following the first loop, prevents the two loops from being fused When the affected variable in the redefinition statement is given a new name and this new name is used in subsequent statements which refer this variable, i e , to statements within the scope of the redefinition, the transformed code becomes as shown to the left, below. On the right is the transformed code following forward scalar substitution.

After Renaming

```
a = 1.0
A(1:N:1) = a
b = 5.0
B(1:N:1) = A(N.1.1) + b
```

After Substitution

```
A(1:N:1) = 1.0
B(1:N:1) = A(N:1:1) + 5.0
```

5.4 Direction Reversal

Direction reversal changes the order in which array references occur within a loop body, without changing the loop increment. For instance, if an array was being produced in an ascending order within a certain loop, direction reversal changes that order so that, in the transformed code, the array is produced in descending order. Note that transformation takes place only within the body of the loop itself; loop control, on the other hand, remains unaffected. The direction reversal transformation is accomplished in essentially the same manner as any other straight forward substitution transformation.

Transformation 5.2 (Direction Reversal) *Reverse the order of array reference within a normalized loop.*

input: A normalized FORTRAN loop, within an eligible loop cluster.

output: A transformed loop in which references to array elements occur in reverse order. Transformation is accomplished by substituting `N - I + 1` for each occurrence of `I` within the loop body, where `N` is the loop upper bound and `I` is the loop index variable.

The following example illustrates how direction reversal is accomplished.

Prior to Reversal

```
      DO 10 I = 1, N
         A(I) = I - 1
10       B(I) = C(N-I+1)
```

After Reversal

```
      DO 10 I = 1, N
         A(N-I+1) = N-I
10       B(N-I+1) = C(N)
```

5.5 Loop Interchange

Loop interchange has been most widely used to increase the vectorization in nested loops in which a loop-carried dependence exists at the innermost level. However, in this instance we use it for an entirely different purpose. Specifically, we use it as a means of providing a complementary reordering of instructions in those cases in which array references vary only according to loop nesting. In this respect, one might consider loop interchange to be to the analysis of multidimensional loops what direction reversal is to the analysis of one dimensional loops, c.f., Chapter 2, Sections 2.4 and 2.5). The transformation itself is straight-forward:

Transformation 5.3 (Loop Interchange) *Interchange the loops within a perfect loop nest.*

input: A perfectly nested FORTRAN loop, within an eligible loop cluster. As a matter of practicality and for the purpose of clarity, we restrict ourselves to the case of two-dimensional loops, such that

```
        DO label I = 1, upper_bound
             DO label J = 1, upper_bound
label.            body
```

where the *upper_bound* for each loop might be different. All array references within the loop body are of the form A(exp_I,exp_J) or A(exp_J,exp_I), where A is an arbitrary array, and exp_I and exp_J are linear expressions in I and J, respectively

output: A transformed loop in which the nesting order is reversed by changing the order of subscripts of all arrays referenced within the loop body. For example, A(exp_I,exp_J) becomes A(exp_J,exp_I) for all A in the loop body, and visa versa.

5.6 Node Splitting

In this section we consider node splitting, a technique which can sometimes increase the effectiveness of cluster analysis. Node splitting has long been used in vectorization to break data dependence cycles, but in cluster analysis it has an additional purpose, as well. Consider, for example, the following loop cluster:

```
         DO 10 I = 1, N
            A(I) = 99.0
10          B(I) = N - I
         DO 20 I = 1, N
20          C(I) = A(I) + B(N-I+1)
```

Since there is no dependency between the two statements in the first loop, the two can be easily placed into distinct loops, rather than within a single loop, as shown. Splitting the two statements allows analysis of the generation order of each array individually, rather than having to consider them as collectively, as one. When the two statements considered together within a single loop, the loops are non-compatible (because two arcs between them are non-compatible), however, when the two statements are separated, each within its own loop, analysis can find a compatible order among the statements, as illustrated by the transformed loops shown next.

After Node Splitting

```
         DO 10 I = 1, N
10          A(I) = 99.0
         DO 15 I = 1, N
15          B(I) = N - I
         DO 20 I = 1, N
20          C(I) = A(I) + B(N-I+1)
```

After Cluster Analysis and Direction Reversal

```
         DO 10 I = 1, N
10          A(I) = 99.0
         DO 15 I = 1, N
15          B(N-I+1) = I - 1
         DO 20 I = 1, N
20          C(I) = A(I) + B(N-I+1)
```

5.7 Loop Fusion

Loop fusion is the first transformation we describe that is applied as a code-improving transformation, rather than as a transformation to facilitate collective analysis. The objective of loop fusion is to to combine loops within a cluster into a single larger loop. As previously noted, the advantages of fusion are many, and the performance benefit substantial (as we will show in Chapter 6). The primary advantages are: *1)* a reduction in loop control overhead as a result of the elimination of all but the first loop control statement, *2)* an opportunity in which to apply other code-improving transformations, such as the substitution transformations previously described and array contraction (described in the next section), and *3)* a single larger loop body, which increases the number of eligible instructions that can be considered for instruction scheduling.

Transformation 5.4 (Loop Fusion) *Fuse two loops that are within a loop cluster*

input: A topological sequence of two normalized FORTRAN loops, of the form

```
              DO label_1 I = 1, upper_bound, 1
label_1           SEQ1
              DO label_2 I = 1, upper_bound, 1
label_2           SEQ2
```

output: A single loop having as its body the statements in the bodies of the loops from which the fused loop is derived, with statements appearing in the same topological sequence as the original loops from which the fused loop was derived, that is

```
              DO label_1 I = 1, upper_bound, 1
                  SEQ1
label_1           SEQ2
```

Notice that although fusion consolidates the statements among compatible clusters of loops within a collection, the fusion process itself takes place on two loops at time. Of course, this in no way affects the performance of the algorithm. Also notice that although loop fusion applies to loops at any particular level of nesting, collective loop analysis is primarily concerned with non-nested loops, the reason being that analysis generally becomes intractable for higher dimensional loops (except for certain special cases, like those described in Chapter 2, Sections 2.5, see 2.6 (pages 23-29). As a simple example of loop fusion, we show next the final code segment from the last section, after the fusion transformation:

```
      DO 10 I = 1, N
         A(I) = 99.0
         B(N-I+1) = I - 1
10       C(I) = A(I) + B(N-I+1)
```

5.8 Array Contraction

The next code improving transformation we describe is called array contraction. Array contraction is the term used by Gao and Sarkar to describe the substitution of scalar variables for subscripted variables within loops which produce and consume arrays [SG91] [4] The fused loop from the previous section is an example of a situation in which such a transformation might be beneficial with the following result:

```
      DO 10 I = 1, N
         a = 99.0
         b = I - 1
10       C(I) = a + b
```

Notice that all but the last array reference, in this case, was able to be replaced. Notice too that contraction has also facilitated scalar forward substitution. Formally, we state the array contraction transformation as

---

Transformation 5.5 (Array Contraction) *Replace with scalar variables all corresponding common subscripted variables which are used to produce/consume array elements within a loop body.*

input: A FORTRAN loop with statements containing subscripted variables which are used to produce and then consume array elements within the same loop iteration, of the form:

```
      DO label I = 1, upper_bound, 1
         A(I) = ...
         .........
         ... = A(I) ...
label    OUT(I) = ..
```

where array A is dead upon exit from the loop, but OUT is live.

[4] Wolfe originally used the term array contraction in the context of removing compiler generated vector temporaries In a general sense, his purpose (to eliminate unnecessary array elements) and the effect of his transformation are the same as those of Gao and Sarkar [Wol89]

output: A transformed loop in which scalar variables have replaced corresponding subscripted variables for arrays which originally produced and consumed elements within the same loop iteration

```
        DO label I = 1, upper bound, 1
           A =

           . = A .
label:     OUT(I) = .
```

5.9 Software Pipelining

In this section we describe a class of loop scheduling techniques collectively known as software pipelining.[5] Software pipelining was originally developed from a technique for scheduling hardware pipelines, but it has been used more recently in compilers for numerous different parallel machines [CGL+89, DHB89, Lam90, RG81, Tou84]. Its original purpose was to reduce the effects of delays between register loads and instruction execution, but perhaps its most important purpose today is to increase instruction-level parallelism. It accomplishes these purposes by replacing instructions within a loop body, with corresponding instructions taken from some later iteration of the loop. Software pipelining is vaguely similar to loop unrolling (see Section 5.10.1, later in this chapter), however, it differs from unrolling in an important respect, and that is, pipelining unrolls iterations for the purpose of finding a pattern of instructions across loop iterations, whereas, loop unrolling aggregates instructions from several iterations. Once a pattern is found as a result of the unrolling, the instructions within the pattern are combined to form the new loop body—a body in which the instructions from across iterations are all concurrently executable. Instructions which occur prior to the pattern then form a loop prolog, and instructions which occur afterwards, an epilog.

As an example of software pipelining we use Aiken and Nicolau's "Optimal Loop Parallelization" technique and the loop shown in Figure 5.1(a) [AN88]. Normally we would expect to apply software pipelining at instruction level; however, for purposes

[5] Weiss and Smith list several possible approaches, and the one we describe in this section is yet another [WS87].

```
      DO 10 I=1, N
         a = N + 1 - I
         b = I - 1
         c = a + b
         d = c + 99
         e = c + b
         t = d + e
         F(I+1) = F(I) + t
10    CONTINUE
```

(a) Prior to Pipelining

a, b				
c				
d, e	a, b			
t	c			
F	d, e	a, b		
	t	c		
	F	d, e	a, b	
		t	c	
		F	d, e	a, b
			t	c
			F	d, e
				t
				F

(b) After Unrolling†

† Letters in Figure (b) correspond to the left-hand-side reference of the statements in Figure (a)

```
a = N+1 - 1
b = 0
c = a + b
d = c + 99
e = c + b
t = d + e
F(2) = F(1) + t

a = N+1 - 2
b = 1
c = a + b
d = c + 99
e = c + b
t = d + e
F(3) = F(2) + t

a = N+1 - 3
b = 2
c = a + b
d = c + 99
e = c + b

a = N+1 - 4
b = 3
```

PROLOG

```
DO 10 I=5, N
   t = d + e
   F(I-1) = F(I-2) + t
   c = a + b
   d = c + 99
   e = c + b
   a = N+1 - I
   b = I - 1
10 CONTINUE
```

BODY

```
t = d + e
F(N+1-1) = F(N+1-2) + t

c = a + b
d = c + 99
e = c + b
t = d + e
F(N+1) = F(N+1-1) + t
```

EPILOG

Figure 5.1: Software-Pipelining[6]

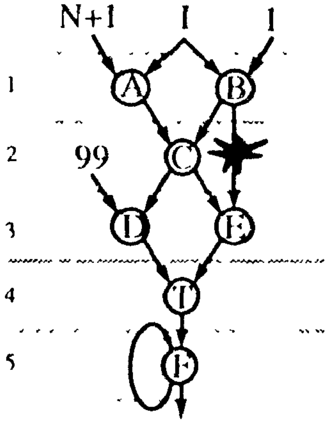
of clarity we use a high-level representation, as shown in the figure.[7] By unrolling iterations of the fused loop, cycle at a time, and scheduling instructions in subsequent iterations as early as possible, as shown in Figure 5.1(b), we soon discover the pattern, indicated within the two horizontal lines. As is shown in this figure, a two cycle pattern appeared after the seventh cycle, beginning with the third iteration. With this information, we are able to then generate the transformed (software pipelined) code shown at the bottom of the figure.

Closer examination of the code in Figure 5.1 reveals that even more parallelism might be exposed were we able to reduce the length of the pattern from two cycles to one. Before attempting to do this, however, let us examine the situation more closely. First, observe that the underlying effect of software pipelining was to change flow dependencies within the original loop to anti-dependencies in the transformed loop. This was accomplished by separating dependent statements among different loop iterations. Whereas, for example, c was flow dependent on a and b in the original loop, (Figure 5.1(a)), it is anti-dependent on these statements in the transformed loop. Likewise, t was flow dependent on d and e in the original loop but became anti-dependent on these latter two statements following transformation.

A second observation that can be made about the pipelining process is that a variable definition cannot be separated from its uses by more than a single iteration. This condition is a direct consequence of every variable being redefined upon every iteration of the loop. If we apply this fact to our analysis of the loop dependence graph shown in Figure 5.2(a), the source of the problem becomes immediately clear. The pattern obtained from unrolling the original code was compressed (as evidenced by the two-cycle pattern) to permit the value of b to be used in statement e; otherwise, the value of b would have been overwritten in the iteration in which it was used in statement c (see the bang in Figure 5.2(a)). To overcome the multiple cycle problem, and yet adhere to the single cycle constraint, requires only that we assign b to a temporary variable bb and use this value for the subsequent computation of e, as shown in Figure 5.2(b). When this is done, the result is a single cycle pattern which now appears after just four cycles. Using this new pattern to derive the body of our loop, we are then able to produce the buffered software pipelined code shown at the

[6] In Figure 5.1(a) we show the fused loop obtained earlier in our example of collective analysis with the last statement split in order to simplify the expression (See Chapter 2, Figure 2.1, page 12).

[7] Since there exists a one-to-one correspondence between the high level statements in our example and their corresponding instruction-level representation, any difference should be of little consequence.



(a) Dependence Graph

a, b				
c, bb	a, b			
d, e	c, bb	a, b		
t	d, e	c, bb	a, b	
F	t	d, e	c, bb	a, b
	F	t	d, e	c, bb
		F	t	d, e
			F	t
				F

† Buffer bb retains the value of b until the value is used by e

(b) After Unrolling

```
a = N+1 - 1
b = 0
c = a + b
bb = b
d = c + 99
e = c + bb
t = d + e

a = N+1 - 2
b = 1
c = a + b
bb = b
d = c + 99
e = c + bb

a = N+1 - 3
b = 2
c = a + b
bb = b

a = N+1 - 4
b = 3
```

PROLOG

```
DO 10 I=5, N-1
    F(I-3) = F(I-4) + t
    t = d + e
    d = c + 99
    e = c + bb
    c = a + b
    bb = b
    a = N+1 - I
    b = I - 1
10 CONTINUE
```

BODY

```
F(N+1-4) = F(N+1-5) + t

t = d + e
F(N+1-3) = F(N+1-4) + t

d = c + 99
e = c + bb
t = d + e
F(N+1-2) = F(N+1-3) + t

c = a + b
bb = b
d = c + 99
e = c + bb
t = d + e
F(N+1-1) = F(N+1-2) + t

a = N+1 - N
b = N - 1
c = a + b
bb = b
d = c + 99
e = c + bb
t = d + e
F(N+1) = F(N+1-1) + t
```

EPILOG

Figure 5.2: Software-Pipelining w/ Buffering

bottom of the figure.

5.10 Incidental Transformations

In this section we describe two additional transformations that are commonly used in production compilers, that affect collective loop analysis, albeit in an indirect way. The first transformation is loop unrolling, and the second is instruction scheduling. We include these transformations because they either form a basis for comparison or their presence can have a pronounced effect upon the quality of our resulting code. We begin with a discussion of loop unrolling.

5.10.1 Loop Unrolling

Loop unrolling, or loop unwinding, as it is sometimes called, has proven itself to be a highly effective transformation in many circumstances [AC71, DH79, Nic88], although its effectiveness is not universal, since it depends upon factors such as the size of the loop being transformed and the architecture upon which the resulting program is to be run. For those situations in which it is applicable, however, loop unrolling has several positive effects: it reduces loop control overhead, it facilitates other transformations (such as wrap-around substitution, described in Section 5.2.2), and it increases the number of instructions within the transformed loop body, increasing the availability of instructions for subsequent instruction scheduling. Among the circumstances in which loop unrolling is apt to be less desirable are: *1)* situations in which the loop body is already large, *2)* situations in which the number of loop iterations is not known at compile time, and *3)* certain situations involving vectorization. From the examples that follow, these effects will become clear. The simplest and most desirable situation, of course, is the one in which the number of loop iterations is a multiple of the unrolling factor, as shown below (with an unrolling factor of 2).

Before Unrolling

```
      DO 10 I = 1, 100
10       A(I) = 99.0
```

After Unrolling

```
        DO 10 I = 1, 120, 3
           A(I) = 99.0
           A(I+1) = 99.0
10         A(I+2) = 99.0
```

If the number of loop iterations is not a multiple of the unrolling factor, two principle alternatives exist: *1)* list the residue statements individually following the loop body or *2)* construct a follow-on loop to execute the residue. By residue we mean statements with indices N-MOD(N,K), where N is the loop size, K is the unrolling factor, and MOD is the modulus function. Since, in practice, unrolling factors are generally small, the first alternative, to list the residue, is almost always least costly, at least in terms of execution time. Had array A in the above code segment been 122, for example, instead of 120, the unrolled loop might have been written as

```
       DO 10 I = 1, 120, 3
          A(I) = 99.0
          A(I+1) = 99.0
10        A(I+2) = 99.0
       A(I+121) = 99.0
       A(I+122) = 99.0
```

When the loop size is not known at compile time, loop unrolling becomes more costly because of the added control necessary to ensure correct loop termination. In this case, an additional comparison operation is performed prior to the corresponding set of statements from each iteration that is unrolled.

```
      DO 10 I = 1, N
         A(I) = 99.0
         IF (I+1) .GT. N GOTO 20
            A(I+1) = 99.0
         IF (I+2) .GT. N GOTO 20
10          A(I+2) = 99.0
20    CONTINUE
```

Another situation in which unrolling can be costly is with respect to codes that are being vectorized. In this case, unrolling can actually degrade performance, rather than improve it. A principle design feature of vector architectures is that they provide special vector instructions to take advantage of long-length vectors, spreading instruction start-up overhead across the entire length of a vector. The problem with loop unrolling is that it tends to inhibit vectorization, causing code which might otherwise be vectorized to run in scalar mode [Don87].

Another area in which loop unrolling can be a problem is with respect to the amount of code generated as a result of the transformation being applied. Of course, loops that are unrolled grow linearly with the number of unrollings. Fortunately, this disadvantage sometimes becomes an advantage, for it is due to this property that loop unrolling finds one of its most important applications: instruction scheduling

5.10.2 Instruction Scheduling

Many instruction scheduling algorithms exist, but most are related to a technique called *list scheduling* [Cof76, Kri90]. The basic notion behind list scheduling is to create an ordered list of tasks, based upon a given set of priorities, and then select tasks from this list, subject to task readiness or resource availability. In the case of instruction scheduling, the tasks are instructions; priorities are primarily determined from information gleaned during a backward scan of a code block dependence graph.

The use of a dependence graph for instruction scheduling is natural because the dependence graph captures, in a concise and easily accessible form, all of the data flow information required for scheduling. In mathematical terms, a dependence graph corresponds to a Hasse diagram in which elemental instructions are the nodes, and data dependencies are represented by arcs. The dependence graph, like a Hasse diagram, shows only the minimum number of arcs necessary to preserve equivalence with the original representation, but unlike a Hasse diagram, the arcs of the dependence graph are weighted by an instruction-latency factor. Another reason for using the dependence graph, besides the fact it contains the right information, is that in many instances the graph could already have been created by the compiler, since dependence graphs have several other code optimization uses as well [ASU86].

For many list scheduling applications, including the present one, a priority list need not actually be constructed. The reason is that the ordering implied by the particular priorities is maintained in a slightly different type of list, called an eligibility list. The eligibility list contains only those instructions which are currently eligible for scheduling, listed according to instruction issue priority. An instruction becomes eligible and, hence, is inserted in the eligibility list, only when the instructions which it is data dependent upon have already been scheduled and, therefore, removed from the candidate list themselves.

Before an instruction is removed from the list, checks are made to avoid scheduling an instruction so as to cause a resource conflict in the hardware pipeline. Conflicts of this type are usually called structural hazards and result in pipeline stalls. When the highest priority instruction in the candidate list cannot be selected because of a hazard, the next highest priority instruction is checked, and so forth, until a hazard free issue can be found. As each successive instruction is removed from the eligibility list, other instructions which were dependent upon the scheduled instruction are added. The instruction insertion-and-selection process then continues until all instructions have been scheduled.

Algorithm VII (Shieh-Papachristou Instruction Scheduling) *Sequence instructions from within a basic block so as to minimize pipeline hazards*

input A dependence graph, or directed acyclic graph (DAG), created from a basic block of instructions, with the instructions being the nodes and data dependencies being the arcs. We refer to this dependence graph as D, a tuple such that $D = (I, A)$, where $i, j \in I$ are instructions, and $A \subseteq I \times I$ are dependence arcs. $c_i \in C_i \subset I$ is a child of i, and similarly, $p_i \in P_i \subset I$ is its parent.

A candidate list $E \subseteq I$ in which instructions that are eligible for scheduling are held, initially $E = \emptyset$.

A ready list $R \subset I$ in which instructions that have already been scheduled are held, initially $R = \emptyset$ also.

A timer variable $t_s = 0$ indicating relative system time.

output: Instructions in R listed according to the following priorities:

1. MAX($load_i$), $\forall i \in I$, where

$$load_i = w_i + \text{MAX}(load_{c_i}),\ \forall c_i$$

and w_i is the execution time, or *weight*, of instruction i,

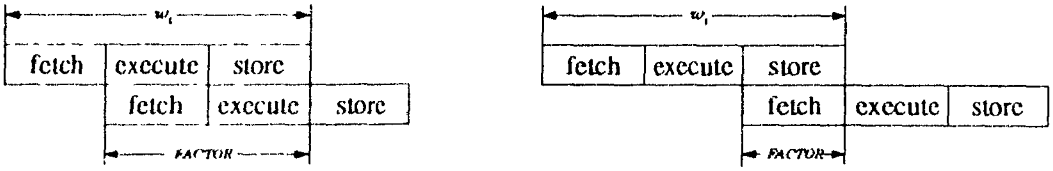
2. MAX(w_i, w_j), if $load_i = load_j$,
3. MAX($|C_i|, |C_j|$), if $w_i = w_j$,
4. MIN($|P_i|, |P_j|$), if $|C_i| = |C_j|$, and
5. MIN($level_i, level_j$), if $|P_i| = |P_j|$, where

$$level_i = 1 + \text{MAX}(level_{p_i}),\ \forall p_i,\ \forall i$$

procedure:

```
.. Assign priorities to instruction nodes
   DO a bottom-up traversal of $D$
      FOR each node $i$ that is visited
         compute $w_i$, $load_i$, $|C_i|$, and $|P_i|$
```

```
        DO a top-down traversal of $D$
            FOR each node $i$ that is visited
                compute $level_i$
    ;; initialize the scheduler
        insert the root nodes into the candidate list $E$, in priority sequence
        set $t_i = 0, \forall e_i \in E$
        move $e_1 \in E$, or top-priority node in $E$, to $R$
        $t_i$++
        DO $\forall c_i \in C_i$
            insert $c_i$ into $E$, keeping $E$ in priority sequence
    ;; assign time slots to the remaining instructions
        WHILE $E \neq \emptyset$ ;; select the highest priority eligible instruction
            selected = false
            $e_i = e_1$
            REPEAT
                $t_i = t_s$
                FOR $p_i \in P_i$
                ;; check for function-unit structural hazards
                    $t_i = MAX(t_p+w_p-FACTOR, t_i)$
                    where
                        $t_i$ is the earliest issue time for $i$,
                        $t_p$ is the issue time of $p_i$
                        $w_p$ is the weight of $p_i$, and
                        $FACTOR$ is.
                            2, if a pipelined function unit uses internal forwarding,
                            1, if a function unit does not use internal forwarding,
                            0, without function-unit pipelining
                IF $t_i = t_s$ ;; there is no structural hazard
                    move $e_i$ to $R$
                    DO $\forall c_i \in C_i$
                        insert $c_i$ in $E$
                    selected = true
                ELSE ;; check the next highest priority instruction for hazards
                    $e_i$++
            UNTIL selected OR $i > |E|$
            $t_s$++
```



(a) Pipeline with Internal Forwarding

(b) Pipeline without Internal Forwarding

Figure 5.3 Determining Function-Unit Pipeline Latency

The particular scheduling algorithm we have described, and the one we use for the performance analysis described in subsequent chapters, is based upon the Shieh-Papachristou method [SP89] This particular method was selected merely because it has been implemented as part of the McGill Compiler-Architecture Testbed and found to be reasonably effective [Muk91] [8]

One important aspect of the Shieh Papachristou method is its capability to schedule floating point instructions on machines which have multiple pipelined function units, both with and without internal forwarding. Internal forwarding allows the result operand of an instruction to be used as an input operand to the follow-on instruction, without the data first being transferred to an externally visible register [HP90]. The way the Shieh-Papachristou algorithm takes internal forwarding into account is by using an instruction latency backoff $FACTOR$, as indicated in the preceding algorithm description There are essentially three situations: *1)* the case of pipelined-function units which employ internal forwarding, *2)* the case of pipelined-function units without it, and *3)* the case of nonpipelined function units The first two cases are illustrated in Figure 5.3. For the function units with internal forwarding, Figure 5 3(a), instruction latency can be reduced by the time required to store the current instruction and fetch the next one; therefore, $FACTOR = 2$. Without internal forwarding only the fetch time for the follow-on instruction is saved, as shown in Figure 5 3(b), so in this case $FACTOR = 1$. Without pipelining the factor is of course zero

An example of Shieh-Papachristou instruction scheduling is shown in Figure 5.4. The segment of source code used for this example is a simple loop containing two

[8] This particular implementation was developed jointly by Chandrika Mukerji and Erik Altman

floating-point arrays (see Figure 5.4(a)). In this example we assume the code produced by our compiler back-end is intended for a DLX computer [HP90].[9] In Figure 5.4(b) we show the dependence graph constructed from the corresponding DLX assembly code, shown in Figure 5.4(c).

The dependence graph in Figure 5.4(b) indicates the minimum flow, anti, and output dependency arcs required to maintain the partial ordering implied by the code, for correct execution. Flow and anti dependencies are determined by comparison of register operands. For example, a flow dependence exists between instructions 1 and 5, with respect to register `f0`, and an anti dependence exists between instructions 1 and 11, with respect to register `r5`. On the other hand, the sequencing of loads and stores is kept the same as in the original assembly code. This is done, in the absence of alias analysis, to ensure output dependencies are honored, e.g., the load in instruction 8 follows the store in instruction 6, just as it does in Figure 5.4(c). Although not essential for correct execution, instructions are placed at their highest level whenever possible, as in the case of instruction 11 which might also have been placed as low as level 8.[10]

In this example there are three specific hazards we would like to overcome as a result of scheduling: *1)* to remove the load stall following instruction 8, *2)* to remove the floating-point stall following instruction 9, and *3)* to fill the branch delay slot following instruction 14. The following description explains how the Shieh Papachristou instruction scheduler proceeds: Assuming that the dependence graph has already been constructed, the first step is to assign a scheduling priority to each instruction. The values of the parameters used in making this determination, along with the resulting priorities, are shown in Figure 5.4(c). Note that the first four parameters described in output section of Algorithm VII (w_i, load$_i$, $|C_i|$, and $|P_i|$) were obtained from a bottom-up traversal of the DAG, whereas, the last parameter (level) was obtained from a top-down traversal.

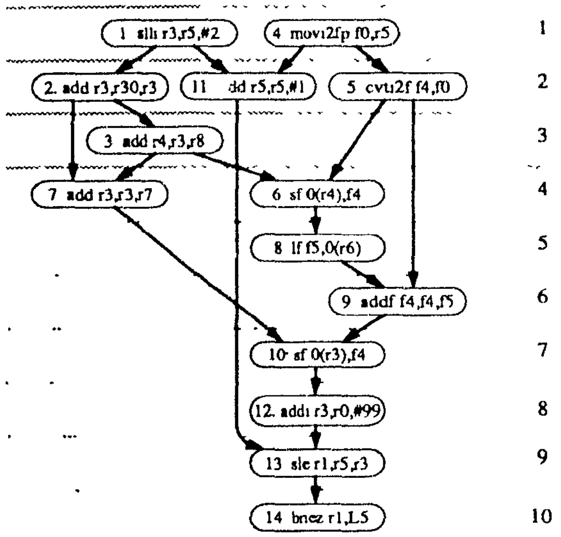
Although the relative priority of each instruction is shown explicitly in Figure 5.4(d), it is actually calculated at the time an instruction is inserted into the candidate list. The steps in the instruction selection process are shown in Figure 5.5. The process begins with the insertion of the root nodes (the highest level nodes) in the

[9] For a brief description of the DLX architecture see Appendix E. In subsequent chapters we use a DLX simulator for performance analysis of collective loop transformations.

[10] To avoid a separate pass over the dependence graph, the current McGill implementation leaves instructions at their lowest level, rather than moving to their highest level, as was done with instruction 11 in Figure 5.4(b).

```
for (i= 0; i<N; i++)
        A(i) = i
        B(i) = A(i) + 99
```

(a) A Simple Code Segment



(b) The Corresponding Dependence Graph

```
 1:  slli r3,r5,#2
 2:  add r3,r30,r3
 3.  add r4,r3,r8
 4:  movi2fp f0, r5
 5:  cvti2f f4,f0
 6:  sf 0(r4),f4
 7:  add r3,r3,r7
 8:  lf f5,0(r6)
 9·  addf f4,f4,f5
10:  sf 0(r3),f4
11:  add r5,r5,#1
12:  addi r3,r0,#99
13:  sle r1,r5,r3
14:  bnez r1,L5
```

(c) Loop Body prior to Scheduling

label	w_i	load$_i$	$\lvert C_i \rvert$	$\lvert P_i \rvert$	level	priority
1	1	13	2	0	1	1
2	1	12	2	1	2	2
3	1	11	2	1	3	5
4	1	12	1	0	1	3
5	1	11	2	1	2	4
6	1	10	2	2	4	6
7	1	6	1	2	4	9
8	2	9	1	1	5	7
9	2	7	1	3	6	8
10	1	5	1	2	7	10
11	1	4	1	2	2	12
12	1	4	1	1	8	11
13	1	3	1	2	9	13
14	2	2	1	1	10	14

(d) Instruction Priority

Figure 5 1· Shieh-Papachristou Instruction Scheduling

eligibility list	selection
{ 1, 4 }	1
{ 2, 4, }	2
{ 4, 5, 11 }	4
{ 5, 3, 11 }	5
{ 3, 11 }	3
{ 6, 7, 11 }	6
{ 8, 7, 11 }	8
{ 9, 7, 11 }	7[1]
{ 9, 11 }	9
{ 10, 11 }	10
{ 12, 11 }	11
{ 12 }	12
{ 13 }	13
{ 14 }	14

[1] avoid structural hazard

(a) Instruction Selection

```
1     slli r3,r5,#2
2     add r3,r30,r3
4     movi2fp f0, r5
5     cvti2f f4,f0
3     add r4,r3,r8
6.    sf 0(r4),f4
8     lf f5,0(r6)
7     add r3,r3,r7
9     addf f4,f4,f5
10    sf 0(r3),f4
11    add r5,r5,#1
12    addi r3,r0,#99
13    sle r4,r5 r3
14    bnez r4,L5
```

(b) Loop Body after Scheduling

Figure 5.5: Shieh-Papachristou Instruction Scheduling Continued

DAG and proceeds in a step-by-step manner through the DAG, selecting at each step the candidate instruction with the highest priority that has no structural hazard, if possible.

The scheduling process for the Simple Code Segment is illustrated in Figure 5.5(a). To begin, instruction 1 has the highest priority, and so it is selected first and removed from the eligibility list (see Figure 5.4(d)). Scheduling instruction 1 makes instruction 2 eligible for selection, and so it is inserted into the candidate list according to its respective priority. Since its priority is higher than instruction 4, the only other eligible instruction, instruction 2 is selected first, making instruction 5 eligible for selection, and so forth. Notice, however, that instruction 9 is not selected when it enters at the top of the candidate list. The reason is a structural hazard caused by the load instruction which immediately precedes it. Since instruction 7 is not affected by the load, and it has second highest priority, it becomes the highest priority eligible instruction and is therefore selected in this case. The final schedule is shown in Figure 5.5(b). As a result of scheduling, two of the three original pipeline hazards that would have occurred using the original schedule have now been alleviated; the third, the branch delay slot, remained unfilled

One final point needs to be made regarding instruction scheduling, and that has to do with the positioning of the scheduler within the compiler itself. Specifically, the order in which code scheduling appears with respect to the register allocator can have a significant impact upon the quality of the resulting code, but whether the scheduler should come before the register allocator, or after, is not clear Because of the dependence of one on the other, however, close interaction between the two is clearly desirable When code scheduling precedes register allocation, the life of variables in registers is apt to be extended by the instruction reordering process; this in turn increases register pressure, perhaps causing register spilling Also, spill code which might be introduced by the register allocator would not be available at the time of scheduling On the other hand, if scheduling follows register allocation, the register allocator reuses registers creating anti dependencies, which in turn places additional constraints upon the extent of reordering the scheduler is able to perform, perhaps causing the resulting code to run less efficiently. In the preceding example, 'nstruction scheduling came last, merely to keep the illustration simple.

Chapter 6

Experiments with Collective Transformation

As suggested in previous chapters, substantial code speedup can be expected as a result of performing collective loop transformations. In this chapter we report test results showing the extent of benefit possible when transformed codes are run on various types of advanced uniprocessor architectures, such as a scalar, a superscalar, and a vector architecture. We also conduct a number of experiments using an scalar processor simulator and cache simulator. For ease of understanding, we separate the results of our experiments into two categories: actual timings and simulations. For each of these categories we examine the performance of a sample collection of compatible loops prior to and following various stages of transformation. We also examine the performance of a collection of non-compatible loops, both for the case of sub-optimal loop partitioning as well as for the case of optimal partitioning. Lastly, we duplicate the preceding series of tests using processor and cache simulators. From these experiments we are able to gain insights into the effects of instruction scheduling and cache memory, results which were not directly observable from timings alone. For both the timing tests and the simulation tests we include alternative transformation strategies, when possible, as a further basis of comparison.

6.1 Uniprocessor Experiments

First we examine the performance of various versions of a compatible loop collection when the codes are run on each of three basic types of uniprocessor: a scalar RISC workstation, a superscalar workstation, and a mainframe with an attached vector processing facility. These machines were respectively a Sun SPARC Server 4/490, an IBM RISC System/6000 (RS/6000), Model 550, and an IBM 3090/VF Model 180J.[1] Following our tests with a compatible collection of loops, we examine a non-compatible collection of loops, to determine the effects of loop cluster partitioning.

The test codes for both sets of experiments were written in FORTRAN and compiled using the native compiler on the host machine. For the Sun 4 the compiler was f77, for the RS/6000, it was xlf, version 1.01, and for the 3090, it was VS FORTRAN, Release 4 (cataloged procedure VSFTCLG). On the Sun Server and RS/6000, each code was executed five-hundred times within an outer loop because the test codes were short and the outer loop was necessary to obtain sufficient timing function resolution, the timing loop was not necessary, and not used, however, for tests on the 3090. For the f77 compiler we used the etime() function (with resolution of one-tenth of a second), for the xlf compiler, we used mtime() (with resolution of one-hundredth of a second), and for VS FORTRAN we used CPUTIME() (with resolution of one microsecond). Extracts of the code listings for these experiments showing exactly how the these timings w re taken are provided in Appendix A.

6.1.1 Tests with Compatible Loop Clusters

In this section we report the results of tests using a compatible set of loops. Our objectives with respect to these tests were to determine *1)* which loop transformations are most susceptible to the basic optimizations provided by the native compiler, *2)* the relative effect from each loop transformation, when considered alone, *3)* the extent of improvement achievable when the various transformations are performed in conjunction with one another, and *4)* the type of uniprocessor architecture (scalar, superscalar, or vector) which benefits most from collective transformation.

Test Suite

For our test suite we used several versions of loop transformations taken from examples in previous chapters. For ease of reference, the original code and various transformed codes are repeated in Figures 6.1 and 6.2, along with other test codes used in this first

[1] In the appendices we describe the salient features of each of these architectures, along with the salient features of their respective compilers.

```
      DO 10 I=1, N
10      A(I) = I
      DO 20 I=1, N
20      B(I) = N - I
      DO 30 I=1, N
30      C(I) = A(I) + B(I)
      DO 40 I=1, N
40      D(I) = C(I) + 99
      DO 50 I=1, N
50      E(I) = C(N+1-I) + B(N+1-I)
      DO 60 I=1, N
60      F(I+1) = F(I) + D(N+1-I) + E(I)
```

(a) Original Loop Cluster

```
      DO 10 I=1, N, 2
        A(I) = I
10      A(I+1) = I+1
      DO 20 I=1, N, 2
        B(I) = N - I
20      B(I+1) = N - I-1
      DO 30 I=1, N, 2
        C(I) = A(I) + B(I)
30      C(I+1) = A(I+1) + B(I+1)
      DO 40 I=1, N, 2
        D(I) = C(I) + 99
40      D(I+1) = C(I+1) + 99
      DO 50 I=1, N, 2
        E(I) = C(N+1-I) + B(N+1-I)
50      E(I+1) = C(N-I) + B(N-I)
      DO 60 I=1, N, 2
        F(I+1) = F(I) + D(N+1-I) + E(I)
60      F(I+2) = F(I+1) + D(N-I) + E(I+1)
```

(b) Original Cluster Unrolled Once

```
      DO 10 I=1, N, 4
        A(I) = I
        A(I+1) = I+1
        A(I+2) = I+2
10      A(I+3) = I+3
      DO 20 I=1, N, 4
        B(I) = N - I
        B(I+1) = N - I-1
        B(I+2) = N - I-2
20      B(I+3) = N - I-3
      DO 30 I=1, N, 4
        C(I) = A(I) + B(I)
        C(I+1) = A(I+1) + B(I+1)
        D(I+2) = A(I+2) + B(I+2)
30      D(I+3) = A(I+3) + B(I+3)
      DO 40 I=1, N, 4
        D(I) = C(I) + 99
        D(I+1) = C(I+1) + 99
        D(I+2) = C(I+2) + 99
40      D(I+3) = C(I+3) + 99
      DO 50 I=1, N, 4
        E(I) = C(N+1-I) + B(N+1-I)
        E(I+1) = C(N-I) + B(N-I)
        E(I+2) = C(N-1-I) + B(N-1-I)
50      E(I+3) = C(N-2-I) + B(N-2-I)
      DO 60 I=1, N, 4
        F(I+1) = F(I) + D(N+1-I) + E(I)
        F(I+2) = F(I+1) + D(N-I) + E(I+1)
        F(I+3) = F(I+2) + D(N-1-I) + E(I+2)
60      F(I+4) = F(I+3) + D(N-2-I) + E(I+3)
```

(c) Cluster Unrolled Three Times

```
      DO 10 I= 1, N
        A(N-I+1) = N-I+1
        B(N-I+1)= I-1
        C(N-I+1) = A(N-I+1) + B(N-I+1)
        D(N-I+1) = C(N-I+1) + 99
        E(I) = C(N+1-I) + B(N+1-I)
10      F(I+1) = F(I) + D(N+1-I) + E(I)
```

(d) Original Cluster Fused

```
      DO 10 I =1, N
        a = N + 1 - I
        b = I - 1
        c = a + b
        d = c + 99
        e = c + b
        f = d + e
10      F(I+1) = F(I) + f
```

(e) Array-Contracted Fused Loop

Figure 6.1: Transformations of a Compatible Loop Cluster

```
C PROLOG
    a = N+1 - 1
    b = 0
    c = a + b
    d = c + 99
    e = c + b
    t = d + e
    F(2) = F(1) + t
    a = N+1 - 2
    b = 1
    c = a + b
    d = c + 99
    e = c + b
    t = d + e
    F(3) = F(2) + t
    a = N+1 - 3
    b = 2
    c = a + b
    d = c + 99
    e = c + b
    a = N+1 - 4
    b = 3
```

```
C BODY
    DO 10 I=5, N
      t = d + e
      F(I-1) = F(I-2) + t
      c = a + b
      d = c + 99
      e = c + b
      a = N+1 - I
10    b = I - 1
```

```
C EPILOG
    t = d + e
    F(N+1-1) = F(N+1-2) + t
    c = a + b
    d = c + 99
    e = c + b
    t = d + e
    F(N+1) = F(N+1-1) + t
```

(a) Software-Pipelined Version

```
C PROLOG
    a = N+1 - 1
    b = 0
    c = a + b
    bb = b
    d = c + 99
    e = c + bb
    t = d + e
    a = N+1 - 2
    b = 1
    c = a + b
    bb = b
    d = c + 99
    e = c + bb
    a = N+1 - 3
    b = 2
    c = a + b
    bb = b
    a = N+1 - 4
    b = 3
```

```
C BODY
    DO 10 I=5, N-1
      F(I-3) = F(I-4) + t
      t = d + e
      d = c + 99
      e = c + bb
      c = a + b
      bb = b
      a = N+1 - I
10    b = I - 1
```

```
C EPILOG
    F(N+1-4) = F(N+1-5) + t
    t = d + e
    F(N+1-3) = F(N+1-4) + t
    d = c + 99
    e = c + bb
    t = d + e
    F(N+1-2) = F(N+1-3) + t
    c = a + b
    bb = b
    d = c + 99
    e = c + bb
    t = d + e
    F(N+1-1) = F(N+1-2) + t
    a = N+1 - N
    b = N - 1
    c = a + b
    bb = b
    d = c + 99
    e = c + bb
    t = d + e
    F(N+1) = F(N+1-1) + t
```

(b) Software-Pipelined Version with Buffer

Figure 6.2: Transformations of a Compatible Loop Cluster - Continued

series of experiments. Among the codes shown are unrolled versions of the primary codes, included as a basis for comparison. Altogether our test suite consisted of seven programs: *1)* the original collection of six loops shown in Figure 6.1(a) (cf. Chapter 2, Figure 2.1, page 12), *2)* the original collection with each loop unrolled once, shown in Figure 6.1(b), *3)* the original collection unrolled three times, Figure 6.1(c), *4)* a single loop created by fusing the original six loops, Figure 6.1(d) (see page 22), *5)* the fused loop transformed by array contraction, Figure 6.1(e) (page 22), *6)* the fused and array-contracted loop after application of software pipelining, Figure 6.2(a), described in Chapter 5 (see Figure 5.1, page 88), and *7)* a second software pipelined loop which employs a buffer, Figure 6.2(b). (Figure 5.2, page 90). Because we were looking for dominant relative effects from the individual transformations, our evaluation objectives were achievable using just the one loop collection. In this respect, we make no claim regarding whether this sample could be part of any actual computation or whether it is representative of codes typically found in practice.

Performance Metric

Our results are reported using a metric for the rate at which output array elements are produced by the various loop transformations, this metric is called *output array elements generated per second*, or *eps*. The eps number represents the rate at which the collective calculation is able to proceed, or in this specific instance, how quickly elements of the output array **F** are produced (see Figures 6.1 and 6.2). The eps number is derived from the slope of the line generated by regressing data points over a range of vector sizes between 60 elements and 3000 elements, at 60 element intervals. The advantage of this metric over a simple average is that it tends to smooth away system anomalies which otherwise might affect the result were the measurements based solely upon a single vector size. Examining a range of vectors sizes can also uncover simultaneous effects, should any be occurring, which could distort the findings, effects such as those which might be caused by cache and memory (we explore these issues in a later section).

Sun SPARC Server 4/490 Results

The results of the first set of tests are shown in Table 6.1, with the results for codes run on the Sun Server in Table 6.1(a). The table includes transformation performance of compiler-optimized and non-optimized versions for both integer and floating point

(a) Performance of Collective Loop Transformations on a Sun SPARC Server 4/490

Phase of	integer		single precision		double precision	
Transformation	f77	f77 -O	f77	f77 -O	f77	f77 -O
original code	177	464	148	302	108	209
unrolled once	241	501	179	341	137	222
unrolled 3 times	269	518	197	296	138	198
fused original	207	568	184	316	134	213
array contracted	561	1792	249	731	183	618
software pipelined	570	1897	245	914	185	642
pipelined w/ buffer	621	1792	237	884	180	680

(b) Performance of Collective Loop Transformations on an IBM RISC System/6000

Phase of	integer		single precision		double precision	
Transformation	xlf	xlf -O	xlf	xlf -O	xlf	xlf -O
original code	245	1665	241	1295	232	1380
unrolled once	304	1945	302	1404	282	1321
unrolled 3 times	373	2099	369	1369	342	1400
fused original	374	2041	355	1061	338	1023
array contracted	707	6736	552	1453	602	2678
software pipelined	730	4604	646	2334	660	3765
pipelined w/ buffer	686	3762	656	2430	659	3747

(c) Performance of Collective Loop Transformations on an IBM 3090/VF, Model 180J

Phase of	integer			single precision			double precision		
Transfrm	OPT(0)	OPT(3)	VEC	OPT(0)	OPT(3)	VEC	OPT(0)	OPT(3)	VEC
original code	309	1330	2583	266	1187	1993	259	816	1698
unrolled once	375	1595	2655	303	1421	1988	286	897	1555
unrolled 3x	401	1789	2066	320	1402	2159	306	975	1275
fused original	338	1401	2181	292	1556	1743	278	848	1409
array contr	916	3751	3915	721	1888	1316	711	1323	1447
softw pipeln	891	3900	2556	726	1999	1229	713	1380	1203
pipeln w/buf	900	2953	1922	726	1778	1097	702	1298	1079

† Results are in thousands of output elements produced per second (eps), based upon regression of vectors within the range 60 3000

‡ IBM 3090 codes were compiled using VS FORTRAN, Release 4

Table 6 1. Performance for a Compatible Loop Cluster

(a) Speedup due to Native-Compiler Optimization

Phase of	Sun, f77 -O			IBM, xlf -O			VS FORTRAN, OPT(3)/VEC†					
Transfrm	int	sgl	dbl	int	sgl.	dbl.	int		sgl		dbl	
original code	2.6	2.0	1.9	6.8	5.1	5.9	1.3	1.9	1.5	1.7	3.2	2.1
unrolled once	2.1	1.9	1.6	6.1	1.6	1.7	1.3	1.7	1.7	1.1	3.1	1.7
unrolled 3x	1.9	1.5	1.4	5.6	3.7	1.1	1.5	1.2	1.1	1.5	3.2	1.3
fused orig.	2.7	1.7	1.6	5.5	3.0	3.0	1.1	1.6	5.3	1.1	3.1	1.7
array contr.	3.2	2.9	3.1	9.5	2.6	2.6	1.1	1.0	2.6	0.7	1.9	1.1
softw. pipeln	3.3	3.7	3.5	6.3	3.6	3.6	1.1	0.7	2.8	0.6	1.9	0.9
pipeln w/buf	2.9	3.7	3.8	5.5	3.7	3.7	3.3	0.7	2.1	0.6	1.8	0.8

† The left data-type column shows speedup for code compiled using "OPT(3)" over "OPT(0), and the right column, speedup using "VEC" over "OPT(3)

(b) Speedup due to Collective Loop Transformation for Compiler-Optimized Code

Phase of	Sun 4/490			IBM RS/6000			IBM 3090/VF‡					
Transfrm	int	sgl.	dbl	int	sgl	dbl	int		sgl		dbl	
unrolled once	1.1	1.1	1.1	1.2	1.1	1.0	1.2	1.0	1.2	1.0	1.1	0.9
unrolled 3x	1.1	1.0	0.9	1.3	1.1	1.0	1.3	0.8	1.2	1.1	1.2	0.8
fused orig	1.2	1.0	1.0	1.2	0.8	0.7	1.1	0.8	1.3	0.9	1.0	0.8
array contr	3.9	2.4	3.0	1.0	1.1	1.9	2.7	1.5	1.6	0.7	1.7	0.8
softw. pipeln	4.1	3.0	3.1	2.8	1.8	2.7	2.9	1.0	1.7	0.6	1.7	0.7
pipeln w/buf	3.9	2.9	3.3	2.3	1.9	2.7	2.2	0.7	1.5	0.6	1.6	0.6

‡ The left data-type column shows speedup of the transformed code over the original code compiled using "OPT(3)", and the right column, the corresponding speedups using "VEC"

Table 6.2. Speedup for a Compatible Loop Cluster

arrays. For ease of analysis, the speedups due to native-compiler optimization and due to the individual source-level transformations, extracted from Table 6.1(a), are shown separately in Table 6.2, along with the corresponding performance ratios derived from similar experiments on the RS/6000 and the 3090. Speedups due to native compiler optimization are shown in Table 6.2(a), and speedups from the successive loop transformations over the performance of the compiler optimized original code are shown in Table 6.2(b).

The Sun performance is interesting for several reasons. First, notice in the three left-most columns of Table 6.2(a), for the Sun 4/490, that native compiler optimization (that is, "f77 -O") resulted, on average, in a two-fold performance improvement over non-optimized code, for the original code, unrolled code, and fused code.

whereas, it resulted in from three to four fold improvement for array-contracted and software pipelined codes, indicating that array-contraction was more susceptible to native compiler optimization than either the original code or the unrolled codes. The reason for this speedup was the elimination of unnecessary store instructions for the intermediate scalar variables which were used in lieu of intermediate array references, as a result of register allocation. Improvement in floating-point code, however, was less than the improvement in the corresponding integer code for the original, unrolled, and fused codes, since the store operations represented a smaller fraction of the the instructions within the loop bodies of the the floating point codes. A small further speedup due to instruction scheduling occurred with respect to optimization of the software pipelined codes. This was because the software pipelining was done at source level, causing array elements to be loaded to register and then used within the same loop iteration. This, of course, would not have occured had the pipelining been done at assembly language instruction level, as normally would be the case.

The improvemert due to f77 optimization, described above, was directly reflected in the performance of the respective loop transformations, see the left-most three columns of Table 6.2(b). The loop-unrolling codes and the loop fusion code, which showed moderate benefit from native compiler optimization, resulted in minimal performance improvement, due to elimination of loop-control overhead and instruction scheduling whereas, the array contracted and software-pipeline l codes which showed substantial speedup from native-compiler optimization, achieved much higher performance than the original compiler optimized code. Also, the combined elimination of array indexing operations and elimination of storage of intermediate variables had considerable impact on the performance of the array-contracted code. The distancing of long latency load operations across loop iterations, combined with native-compiler instruction scheduling, increased the speedup from software pipelining, and as a result, in absolute terms, software pipelining achieved the most speedup, 4.1, cf. the first column of Table 6.2(b), with the software-pipelined code for integer arrays achieving the highest performance overall, 1897 eps, cf. the second column of Table 6.1(a).

IBM RISC System/6000 Results

In Table 6.1(b) are the results of executing the test codes on the RS/6000, with the corresponding speedups for the various transformations shown in the middle three columns of Tables 6.2(a) and (b). Again, Table 6.2(a) contains the speedups due to native compiler optimization, in this case "`xlf -O`", and Table 6.2(b) contains the speedups from successive loop transformations over the performance of the original code, for code compiled using "`xlf -O`"

Two observations are immediately apparent from these experiments *1)* the wide variance in the results from codes that use integer type arrays to the results from codes that use floating-point arrays and *2)* the wide difference in results on the RS/6000 compared with results on the Sun

First, notice that the spread in improvement due to optimization by the native xlf compiler was relatively large, compared to the Sun f77 optimizations, (cf. Table 6.2(a)), with a five- to six-fold improvement resulting from compiler optimization in the case of codes having integer arrays, and three- to four-fold improvement from optimization of codes having floating-point arrays The higher effectiveness of xlf optimization on the RS/6000 compared with f77 optimization on the Sun was due to register allocation and the usage of hardware assisted auto incrementing loop control within the loop bodies In the case of the array contracted code for integer arrays, the xlf compiler reduced the number of instructions within the loop body from 39 to 10 with a corresponding reduction in the estimated number of cycles from 51 to just 9 Because of the additional data-conversion instructions required, less improvement occurred with the codes for floating-point, especially with respect to the single precision codes.

The improvement from loop transformation, although much higher in absolute terms, was lower in relative terms than the corresponding Sun results, although as before, array contraction resulted in the greatest improvement overall, cf. Table 6.2(b). The smaller speedups were because greater instruction level parallelism in the RS/6000 architecture benefited all loop transformations equally, making the improvement due to collective transformation a smaller proportion of the overall gain Loop unrolling and loop fusion produced about the same speedup over the original code as they did on the Sun, but the speedup from array contraction was less on the RS/6000. Single-precision floating-point results, in general, were especially low when compared to the integer and double-precision floating-point results (see the middle column of Table 6.2(b)) This is because floating-point calculations on the RS/6000 are done in double-precision, causing single-precision floating-point calculations to be converted.

Notice that array contraction failed to significantly improve codes on the RS/6000 for single-precision floating-point arrays. This was because the xlf compiler converted, stored, and reloaded intermediate variables, within the loop body, rather than using registers, as in the case of the corresponding integer codes. On the RS/6000, single-precision software-pipelined codes too performed less well than the corresponding integer or double-precision floating-point codes And, for all three data types, the effects of software pipelining were less consistent; in some instances improving performance while in other instances, decreasing it. This was due to the mix of instructions and the ability of the instruction scheduler to take advantage of the machine's multiple function units Overall, collective loop transformation resulted in less improvement on the RS/6000 than on the Sun, however, in absolute terms, the RS/6000 was three- to six times faster with loop unrolling codes and two- to three-times faster than the Sun for the remaining codes

IBM 3090/VF, Model 180J, Results

The results of running the preceding experiments on the 3090 are shown in Table 6 1(c), and the corresponding speedups from using VS FORTRAN optimization/vectorization and the speedups for each successive loop transformation over the performance of the optimized/vectorized original code are shown on the right-hand side of Tables 6 2(a) and (b) Notice that for each data type shown in the two tables there are two columns this time, instead of one. In the left-hand column for each data type heading in Table 6 2(a) are the speedups due to scalar optimization. i.e., from using "OPT(3)", over the results obtained from non-optimized code, using "OPT(0)", and in the right-hand column for each data type heading are the additional speedups obtained by vectorization, or "VEC", over that obtained by using OPT(3).[2] In Table 6.2(b) are the respective speedups due to the successive loop transformations over the performance of the original code (in the left-hand columns) for optimized code, or using OPT(3), and (in the right-hand columns) for vectorized code, or using VEC The statistics in these tables illustrate several differences between the performance of the 3090 and the other two machines tested

First, examine the speedup of VS FORTRAN using OPT(3) (optimization without vectorization), or the left-hand column under each data-type heading in Table 6.2(a). Notice that the speedup from native-compiler optimization was greater on the 3090 than it was on the Sun, except for the double-precision floating-point array-contracted code and two software-pipelined codes (shown in the lower half of the second to the last column) Also notice there exists an inverse relationship between VS FORTRAN optimization on the 3090 and f77 optimization on the Sun, shown

[2] Vectorization, or VEC, assumes optimization at the level of OPT(3)

in the left-most three columns of the Table (a). Whereas on the Sun, the unrolled code and fused code were improved less by native compiler optimization than the contracted code and pipelined codes, the corresponding codes on the 3090 were improved more. These differences are due to the fact that the 3090 allows arithmetic and logic instructions to have memory operands, and VS FORTRAN compiler takes advantage of this capability during optimization, doing away with a number of load instructions. The SPARC, on the other hand, is a load/store architecture, and this type of transformation does not apply. At the same time, improvement from VS FORTRAN optimization on the 3090 was generally less than xlf optimization for the RS/6000, because the 3090 lacks the sophisticated hardware support for instruction level parallelism which exists on the RS/6000, which the xlf compiler was able to use (specifically, the auto-indexing feature). Lastly, unlike the f77 and xlf compilers, the VS FORTRAN compiler improved unrolled code equally as much as it did the original code. Again, the primary reason for this marginal improvement, over the improvement achieved by native-compiler optimization on the other two machines, was the use memory operands.

With respect to the vectorized codes, a number of additional characteristics stand out. First, vectorization had a pronounced effect upon the speedup of the original code, increasing performance by as much as 8.1 (for integer arrays). This was the highest speedup for any code, see Table 6.1(c). However, it was not the best performance, the array-contracted integer code performed better, achieving a rate of 3915 eps (compared to a rate of 2583 eps for the original code).

Table 6.1(c) indicates that, for single-precision arrays, the code that was unrolled three times performed best (2159 eps), but this was not actually the case, closer analysis of the data showed that the original code performed better.[3] Likewise, for double-precision arrays, the original code was best (1698 eps). In spite of the deviations, the statistics in the table clearly indicate that unrolling reduced the effect of vectorization more times than not, and loop fusion had a similar effect in either case, reducing performance by as much as twenty percent. The follow-on transformations (array contraction, and software pipelining) also achieved lower performance, especially for the software-pipelined codes that employed the instruction buffer, since array contraction virtually eliminated the vectorizable portion of these codes causing the respective codes to execute at approximately the same rate as the corresponding scalar code compiled with `OPT(3)`, again, see Table 6.1(c).

[3] The discrepancy in the numbers was due to rounding error in the regression calculation

In absolute terms, the 3090 out-performed the RS/6000 by a significant margin on the original code, both for integer and floating-point arrays, cf. Tables 6.1(b) and (c). The 3090 was 1.6 times faster for integer arrays, 1.5 times faster for single-precision floating point arrays, and 1.2 times faster for double-precision floating-point arrays. However, transformed versions of the code run on the RS/6000 run better overall, by an even wider margin For example, the buffered software-pipelined code on the RS/6000 ran 2.0 times faster than vectorized code on the 3090 for integer arrays, 2.2 times faster for single precision code, and 3.5 times faster for double-precision code More impressive is the fact that the xlf compiler produced the fastest code for each data type tested, but only because of collective loop transformation: 6736 eps for integer codes, compared with 3915 on the 3090, 2430 eps for single-precision floating-point codes, compared with 2159 eps, and 3765 eps for double-precision floating-point codes, compared to 1698 eps. On the other hand, had collective loop transformations not been used, the 3090 would have out-performed the RS/6000 in every instance, clearly showing that collective loop transformations can make a crucial difference in performance

6.1.2 Tests with a Non-Compatible Cluster of Loops

In this section we describe the results of loop-partitioning experiments using a collection of non-compatible loops Our objectives in this case were to *1)* measure the effect of efficient (optimal) loop partitioning over naive (sub-optimal) partitioning, *2)* isolate the effects of loop fusion and array contraction when applied at various stages of collective transformation. *3)* determine the effect of native compiler optimization upon our source-level transformations, and *4)* compare the effectiveness of loop transformations on computers having architectures which support different degrees of instruction-level parallelism Except for the first objective, these objectives are essentially the same as those described for the experiments in the previous section.

Test Suite

To illustrate the impact of partitioning strategy upon the performance of a non-compatible collection of loops, we used the loop collection described in Chapter 3, Figure 3.8 (page 19). The particular FORTRAN code for this collection is shown in Figure 6.3, along with several transformed versions of the code.

The collection contains two sets of non-compatible loops, the first involving loops

```
      DO 10 I=1, N
10       A(I) = I
      DO 20 I=1, N
         B(I) = A(I) * 2 + 3
20       C(I) = B(I) + 99
      DO 30 I=1, N
30       D(I) = A(N-I+1) + 6
      DO 40 I=1, N
40       E(I) = B(I) + C(I) * D(I)
      DO 50 I=1, N
         F(I) = E(I) * 4 + 2
50       G(I) = E(I) * 8 - 3
      DO 60 I=1, N
60       H(I) = F(I) + G(I) * E(N-I+1)
```

(a) Original Loop Cluster

```
      DO 10 I=1, N
10       A(I) = I
      DO 20 I=1, N
         b = A(I) * 2 + 3
         B(I) = b
20       C(I) = b + 99
      DO 30 I=1, N
30       D(I) = A(N-I+1) + 6
      DO 40 I=1, N
40       E(I) = B(I) + C(I) * D(I)
      DO 50 I=1, N
         e = E(I)
         F(I) = e * 4 + 2
50       G(I) = e * 8 - 3
      DO 60 I=1, N
60       H(I) = F(I) + G(I) * E(N-I+1)
```

(b) Array-Contracted Original Cluster

```
      DO 10 I=1, N
         A(I) = I
         B(I) = A(I) * 2 + 3
10       C(I) = B(I) + 99
      DO 20 I=1, N
         D(I) = A(N-I+1) + 6
         E(I) = B(I) + C(I) * D(I)
         F(I) = E(I) * 4 + 2
20       G(I) = E(I) * 8 - 3
      DO 30 I=1, N
30       H(I) = F(I) + G(I) * E(N-I+1)
```

(c) Fused Loops from Naive Partitioning

```
      DO 10 I=1, N
         a = I
         b = a * 2 + 3
         A(I) = a
         B(I) = b
10       C(I) = b + 99
      DO 20 I=1, N
         d = A(N-I+1) + 6
         e = B(I) + C(I) * d
         E(I) = e
         F(I) = e * 4 + 2
20       G(I) = e * 8 - 3
      DO 30 I=1, N
30       H(I) = F(I) + G(I) * E(N-I+1)
```

(d) Array-Contracted Naive Partitioning

```
      DO 10 I=1, N
10       A(I) = I
      DO 20 I=1, N
         B(I) = A(I) * 2 + 3
         C(I) = B(I) + 99
         D(I) = A(N-I+1) + 6
20       E(I) = B(I) + C(I) * D(I)
      DO 30 I=1, N
         F(I) = E(I) * 4 + 2
         G(I) = E(I) * 8 - 3
30       H(I) = F(I) + G(I) * E(N-I+1)
```

(e) Fused Loops from Efficient Partitioning

```
      DO 10 I=1, N
10       A(I) = I
      DO 20 I=1, N
         b = A(I) * 2 + 3
         c = b + 99
         d = A(N-I+1) + 6
20       E(I) = b + c * d
      DO 30 I=1, N
         e = E(I)
         f = e * 4 + 2
         g = e * 8 - 3
30       H(I) = f + g * E(N-I+1)
```

(f) Array-Contracted Efficient Partitioning

Figure 6.3: Transformations of a Non-Compatible Loop Cluster

1 through 4 (identified by labels 10 through 40) and the second involving loops 1 through 6 (labels 10 through 60) Were this collection of loops part of some source input, an optimizing compiler might perform any of the source-to-source transformations shown in Figure 6.3 *1)* forgo collective transformation altogether, leaving the source as shown in Figure 6.3(a), *2)* perform array contraction on the original cluster without performing fusion, as shown in Figure 6.3(b) (contraction actually occurs in only two of the six loops in this figure: loops 2 and 5), *3)* naively partition the collection, possibly cutting more arcs than necessary and then perform loop fusion, as in Figure 6.3(c), *4)* naively partition the loops and perform array contraction within each partition, Figure 6.3(d), or alternatively, *5)* efficiently partition the graph and stop, as in Figure 6.3(e), or as we would hope, *6)* efficiently partition the graph and perform array contraction within the loop bodies of the respective loops formed by the partitions, as shown in Figure 6.3(f). Notice that in some instances the code was rewritten to assist the compiler with array-element recognition The particular modification is shown in the array-contracted original collection where scalar references are used for array elements to avoid redundant array accesses, refer to Figure 6.3(b).

Test Results

The results of partitioning based upon timings taken on the Sun 4/490, the RS/6000, and the 3090 are shown in Tables 6.3(a), (b), and (c), respectively. For each of these tables, the format is the same as it was for the corresponding tables described in Section 6.1.1 As before, the tables show both optimized and non-optimized versions for integer, single-precision floating-point, and double-precision floating-point arrays Also as before, the results are reported in *eps*. In this particular instance, however, the eps metric indicates the rate at which elements of the output array H were produced, based upon a regression of execution times for vector lengths between 60 and 3000 elements, measured at 60-element intervals. Finally, to assist in evaluating the results, we show in Table 6.4 the corresponding speedups, first, due to native-compiler optimization, viz Table 6.4(a), and second, due the various loop transformations viz Table 6.4(b), for each array data type tested Lastly, since the test codes used in these experiments also were short, each code was executed five-hundred times within an outer loop, to obtain sufficient timing-function resolution.

The results in Table 6.3, and Table 6.4(b), show the importance of efficient partitioning, with efficient partitioning and transformation resulting in significantly better

(a) Performance of Loop-Cluster Transformations on a Sun SPARC Server 4/490

Phase of	integer		single precision		double precision	
Transformation	f77	f77 -O	f77	f77 -O	f77	f77 -O
original loop collection	90	165	117	202	86	135
array-contracted original	97	168	117	217	82	136
fused naive partition	103	171	137	210	96	133
array-contr. naive part.	124	191	136	242	95	151
fused efficient partition	113	178	137	215	97	135
array-contr. efficient part.	149	225	151	329	116	220

(b) Performance of Loop-Cluster Transformations on an IBM RISC System/6000

Phase of	integer		single precision		double precision	
Transformation	xlf	xlf -O	xlf	xlf -O	xlf	xlf -O
original loop collection	195	939	201	1152	195	1202
array-contracted original	191	933	197	1142	191	1201
fused naive partition	245	1089	253	1101	239	1239
array-contr. naive part.	259	1109	247	1139	248	1325
fused efficient partition	247	1198	248	1306	252	1316
array-contr. efficient part.	322	1542	332	1363	329	2494

(c) Performance of Loop-Cluster Transformations on an IBM 3090/VF, Model 180J

Phase of	integer			single precision			double precision		
Transfrm	OPT(0)	OPT(3)	VEC	OPT(0)	OPT(3)	VEC	OPT(0)	OPT(3)	VEC
orig collection	228	665	2285	208	606	2036	203	529	1511
contr original	238	681	2188	215	625	1996	208	548	1500
fused naive	261	746	2523	231	630	2151	223	526	1474
contr naive	302	876	2615	276	684	2235	262	583	1718
fused efficient	261	701	2504	232	627	2157	226	531	1681
contr efficient	415	859	3268	346	733	2834	335	669	2913

† Results are in thousands of output elements produced per second (eps), based upon regression of vectors within the range 60 3000

‡ IBM 3090 codes were compiled using VS FORTRAN, Release 4

Table 6.3: Performance for a Non-Compatible Loop Cluster

(a) Speedup due to Native-Compiler Optimization

Phase of	Sun, f77 -O			IBM, xlf -O			VS FORTRAN, OPT(3)/VEC†					
Transfrm	int	sgl.	dbl.	int.	sgl.	dbl	int.		sgl		dbl.	
original code	1.8	1.7	1.6	4.8	5.7	6.2	2.9	3.4	2.9	3.1	2.6	2.9
contr original	1.7	1.8	1.7	4.9	5.8	6.2	2.9	3.2	2.9	3.2	2.6	2.7
fused naive	1.7	1.5	1.1	4.4	4.4	5.2	2.9	3.4	2.7	3.1	2.1	2.8
contr naive	1.5	1.8	1.6	4.3	4.6	5.3	2.9	3.0	2.5	3.3	2.2	2.9
fused efficient	1.6	1.6	1.4	4.9	5.3	5.3	2.7	3.6	2.7	3.4	2.3	3.2
contr efficient	1.5	2.2	1.9	4.8	4.1	7.6	2.1	3.8	2.1	3.9	2.0	1.4

† The left data-type column shows speedup for code compiled using "OPT(3)" over "OPT(0), and the right column, speedup using "VEC" over "OPT(3)

(b) Speedup due to Loop-Cluster Transformation for Compiler-Optimized Code

Phase of	Sun 4/490			IBM RS/6000			IBM 3090/VF‡					
Transfrm	int.	sgl	dbl	int.	sgl.	dbl	int.		sgl.		dbl.	
contr original	1.0	1.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
fused naive	1.0	1.0	1.0	1.2	1.0	1.0	1.1	1.1	1.1	1.1	1.1	1.0
contr naive	1.2	1.2	1.1	1.2	1.0	1.1	1.3	1.1	1.3	1.1	1.3	1.1
fused efficient	1.1	1.1	1.0	1.3	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1
contr efficient	1.1	1.6	1.6	1.6	1.2	2.1	1.8	1.4	1.7	1.4	1.7	1.9

‡ The left data-type column shows speedup of the transformed code over the original code, compiled using "OPT(3)", and the right column, the corresponding speedups, using "VEC"

Table 6.4 Speedup for a Non-Compatible Loop Cluster

performance in almost all instances. In addition, a number of other important conclusions can be reached with regard to the transformations themselves. In most cases, these conclusions reinforce findings from the previous experiments reported in Section 6.1.1.

First, notice that the speedups from native-compiler optimization, shown in Table 6.4(a), were as much as one-third lower for this set of experiments than they were for the last one, cf Table 6.2(a) (page 107). The lower performance was due to the fact that the codes used in this case contained larger expressions and a higher proportion of straight-line integer operations than the codes used in the previous experiments, and as a result, the portion of the code which was actually affected by native-compiler optimizations represents a smaller proportion of the corresponding non-optimized code. Recall, the primary cause of improvement due to optimization was the elimination of unnecessary load and stores between each use of a variable, and these remained roughly the same; whereas the number of arithmetic operations increased. Consequently, the speedups due to native compiler optimization of the fused codes and array-contracted codes were about the same as they were for the original code. The lower speedup due to compiler optimization was likewise manifest in lower pe:formance of the respective loop transformations, especially for the fused codes; see Table 6.1(b). In this latter case, the reason is that the eliminated loop control overhead, or the primary source of speedup, was an even smaller proportion of the original code than it was for the previous experiments, shown in Table 6.2(b).

Next notice, in Table 6.1(b), that the speedups from the various loop transformations occurred uniformly across the three architectures. The speedups for codes that used loop fusion alone, for example, were close to the speedups of the respective original codes for each machine. Likewise, the speedups of the contracted codes improved by roughly the same amount over the speedups of the respective original codes. For the few instances in which significant differences in performance did occur, the differences applied uniformly across all transformations on the particular machine with none of the transformations appearing to be more or less susceptible to any particular architectural effect than any other. For the RS/6000, for example, the results with floating-point arrays and those for integer arrays were closer together than they were for the last set of experiments, and for the 3090, vectorization improved code performance more this time than it did during the previous experiments. These differences again reflect the fact that these particular codes contained larger expressions and a higher proportion of straight-line integer operations than before. As a consequence, less opportunity existed for instruction level parallelism in the code that used integer arrays, and integer operations comprised a larger proportion of the codes that used floating-point arrays.

Next we examine the performance effects from applying each of the individual loop transformations shown in Table 6.4(b), refer to Figure 6.3(b)–(f). First, notice that little, if any, speedup occurred as a result of applying array contraction to the original loop, as shown Figure 6.3(b), see the first row in Table 6.3(b). This was because no arrays were actually eliminated by the contraction. Instead, a few memory references were replaced by register references, representing only a small proportion of the overall computation. Nevertheless, the transformation provides a basis for comparing test transformations, giving an indication of the importance of this particular effect, vis á-vis Figures 6.3(d) and (f).

When the loop cluster was naively partitioned to allow loop fusion, as shown in Figure 6.3(c), the speedup was again small because all that was eliminated in this case was loop-control overhead which too was small compared to the size of the overall computation, cf. row two in Table 6.4(b) The efficiently partitioned loops, Figure 6 3(e), also showed little improvement for the same reason, cf third row of Table 6 4. In both of the preceding cases, the effect of transformation was merely to reduce the number of loops from six to three Recall, a potential benefit from loop fusion is to increase the size of the corresponding loop bodies, so as to increase the pool of instructions available for instruction scheduling However, in this particular instance, the larger fused loops made little difference. In spite of the lack of improvement, loop fusion provided an important advantage by facilitating array contraction

The speedups from both loop fusion and array contraction together were far greater than the speedups from any of the other tests described with respect to the non-compatible loops, see Figures 6 3(d) and (f), and Table 6 4(b) Part of the reason was, of course, the incremental improvement provided by the prior fusion, but the main reason was the elimination of an array Notice that in the second loop nest in Figure 6 3(d), the D array was eliminated entirely and replaced by the scalar variable d. The last test code, for an array-contracted efficient partitioning, Figure 6.3(f) showed much greater improvement approaching a two-fold improvement on the RS/6000 for double-precision floating point arrays, cf Table 6 4(b) The reason is obvious five of the eight original arrays were eliminated and replaced by scalar variables, and of the three remaining arrays, one was the output array H and the other two (A and E) correspond to the minimum number of nonfusible arrays.

One additional point remains: collective loop transformations were necessary to achieve best performance for all three machines (including the 3090). In absolute terms, the 3090 outperformed the other two machines by a wide margin For codes that used integer arrays, the 3090 was fourteen times faster than the Sun, and eight-to twelve-times faster for floating-point numbers At the same time, it was more than twice as fast as the RS/6000 in several instances This is especially significant since the RS/6000 outperformed the 3090 during the first set of experiments. From these results it appears that there is also good reason to perform collective transformations on codes for vector machines.

6.2 DLX Simulation

In this next section we examine the effects of collective loop transformations when test codes were run on a RISC processor simulator and a cache simulator. The processor simulator used in this case was an enhanced version of the DLX simulator from Stanford, and the cache simulator was the *dinero* simulator from Berkeley [HM90, HS84]. Although DLX is a simulated architecture, it provides all of the essential features one might expect from a RISC processor, and it includes the instrumentation features required for our analysis. For simple instruction sequences, that is for sequences without special system calls, such as `printf()`, the DLX simulator interprets and then executes each instruction as though it were executed on a real machine. In doing so, however, it assumes the presence of an infinite warm cache in which all instructions are present, i.e., there are no cache misses, and all data and instructions are accessible in constant time (see Appendix E). The specific DLX machine configuration used for our experiments consisted of the base architecture with the following additional floating-point hardware: an add/subtract unit having a latency of 2 cycles, a multiplier unit having a latency of 5 cycles, and a divider unit having a latency of 19 cycles. For our cache simulations, we assumed an 8K-byte instruction cache and a 64K-byte data cache, both being 4-way set associative with 128 byte lines and using a least-recently-used (LRU) line-replacement policy.

The same source codes were used for the simulation experiments described in this section as were used in our earlier experiments, however, for these particular experiments the sources were converted from FORTRAN into C to permit compilation by an upgraded version of the *dlxcc* compiler (version 2). The dlxcc compiler generates the DLX assembly code which is then used as source input to the simulator. Because our test codes were simple, the particular source language used in this instance had little impact on the test results.[4] This was because, for the most part, there existed a one-to-one correspondence between statements within corresponding test codes for each of the two languages.

One final factor differentiates the following simulation experiments from the timing experiments described in the previous sections, and that is, several of the codes used in the following experiments were compiled twice, once with instruction scheduling and once without. The particular instruction scheduler used by the compiler was based upon a "postpass" version of the Shieh-Papachriston algorithm, Algorithm VII, described in Chapter 5 (page 94).[5] The only structural hazard considered by

[4] This fact was verified by comparing timing results of FORTRAN and C versions of the test codes used in the previous experiments.

[5] As indicated in Chapter 5, Chandrika Mukerji and Erik Altman jointly implemented the scheduler used for these experiments.

the scheduler, and the only hazard enforced by the DLX simulator itself, was a one-cycle delay between the time a value is loaded into register and the time it is used. The simulator also delays the effect branch instructions by one cycle, as explained in Appendix E. To account for this delay, the dlxcc compiler merely inserts a NOP instruction following each branch instruction.[6]

6.2.1 Tests with Compatible Loop Cluster

The objectives for our first series of experiments were again essentially the same as for the compatible-collection timing experiments described in Section 6.1.1, specifically we wanted to determine: *1)* the susceptibility of each transformation to native-compiler optimization, *2)* the relative improvement of each transformation, when considered alone, and *3)* the combined effect of transformation. In addition, we wanted to determine the impact of instruction scheduling. This later objective was important because our experiments were focused toward the effects of source-level (or source-to-source) transformations, and instruction scheduling was expected to be an important factor in the outcome of our tests. Lastly, we wanted to measure the effects of register and cache usage.

The actual codes used for the first series of simulations were C versions of the codes shown in Figures 6.1 and 6.2 (pages 103–104), and the number of loop iterations (hence, vector size), "`N`", used for these simulations was 100.[7] Again, the basis for these experiments was the compatible collection of loops shown in Figure 6.1(a). For each test code we measured performance under three different optimizing configurations. *1)* without optimization, *2)* without optimization but using register variables for intermediate values, and *3)* with full optimization. By full optimization we mean "`dlxcc -O`". The following sections describe results, first for integer-defined arrays, then for arrays of single-precision floating-point numbers, and finally for arrays of double-precision floating-point numbers. As in our previous experiments, immediate values were integers, regardless of the data type of the array elements.

[6] The effect of adding NOPs to fill the branch-delay slot is reported in the analysis that follows.

[7] The actual C versions of these test codes are included with the source listings provided in Appendix A.

Tests with Integer Arrays

For experiments using integer-type array elements, the test results are shown in Table 6.5, with the results of non-optimized versions of the initial transformations in Table 6.5(a). Also, as in the case of our previous experiments, we have summarized the speedups due to compiler optimization and the speedups due to the successive transformations, in Table 6.6(a) and (b), respectively.

The most significant point with respect to these statistics is the dramatic increase in performance due to array contraction. Table 6.6(b) shows a three fold speedup for the non-optimized array-contracted code over the original code, a five fold speedup for the corresponding codes in which scalar variables were defined as register variables, and a four-fold speedup for optimized codes i.e., codes compiled with "dlxcc -O". The reason for these speedups can be seen from Tables 6.5(a)-(c). Notice that for non-optimized code, shown in Table 6.5(a), the number of loads dropped from 2601, for the fused code, to 1101, for the array-contracted code, and the number of instructions dropped from 12407 to 4407, both as a result of replacing array references (which were stored in memory) by scalar references and by eliminating the respective load instructions and indexing instructions. For the array contracted codes in which register variables were used (see Table 6.5(b)) and for the fully optimized code (Table 6.5(c)), the impact was an eight-fold reduction in the number of loads and a four- to five-fold reduction in the number of instructions.

The tables which comprise Table 6.5 expose another factor was not so apparent from the timing experiments described earlier in this chapter, and that is the importance of keeping live data within the registers once this data is there. In this case merely defining scalar variables as register variables reduced the number of loads from 3606 to 800, and it reduced the number of load stalls from 1700 to 500. The effect was to eliminate the storing and reloading of intermediate scalar values, a transformation which was accomplished anyway as part of register allocation during usual code optimization. Recognizing and eliminating common sub-expressions which were part of the array references eliminated an additional 300 loads and 200 stalls, as reflected in the difference in number of loads and stalls between the original code and the fused code, cf. Table 6.5(c).[8] And lastly, applying array contraction to the fused code, eliminated yet an additional 400 loads (hence, 400 instructions).

Among the test results included in Table 6.5(c), for compiler optimized codes, are

[8] However, common sub-expression elimination was incomplete; had it been complete, array contraction would not have been necessary.

(a) Performance of Collective Loop Transformations using dlxcc without Optimization

Phase of	integer		load		instr	run time
Transformation	loads	stores	stalls	NOPs	count	(in cycles)
original code	3606	1206	1700	1211	15947	17647
unrolled once	3006	906	1850	611	12997	14847
unrolled 3 times	2706	756	1975	311	11522	13497
fused original	2601	701	1900	201	12407	14307
array contracted	1401	701	900	201	4407	5307
software pipelined	1477	797	980	193	4514	5494
pipelined w/ buffer	1571	896	1075	191	4691	5766

(b) Performance of Collective Loop Transformations with Register Variables, but without dlxcc Optimization

Phase of	integer		Load		instr.	run time
Transformation	loads	stores	stalls	NOPs	count	(in cycles)
original code	800	600	500	1211	12535	13035
unrolled once	800	600	500	611	10485	10985
unrolled 3 times	800	600	500	311	9460	9960
fused original	800	600	500	201	10505	11005
array contracted	100	100	100	201	2505	2605
software pipelined	100	100	100	193	2441	2541
pipelined w/ buffer	100	100	100	191	2525	2625

(c) Performance of Collective Loop Transformation using dlxcc -O

Phase of	integer		load		instr.	run time
Transformation	loads	stores	stalls	NOPs	count	(in cycles)
original code	800	600	500	606	7949	8449
instr-scheduled orig	800	601	200	606	7950	8150
unrolled once	750	600	500	306	6159	6659
unrolled once, sched	750	601	200	306	6160	6360
unrolled 3 times	725	600	500	156	5733	6233
unrolled 3x, sched	725	601	150	156	5734	5884
fused original	500	600	300	101	3611	3911
fused orig, sched	500	601	100	101	3612	3712
array contracted	100	100	100	101	1808	1908
contracted, sched	100	101	100	101	1809	1909
software pipelined	97	100	97	97	1849	1946
pipelined, shed	96	99	96	97	1850	1946
pipelined w/ buffer	96	100	96	96	1940	2036
pipln w/ buf, sched	96	100	96	96	1942	2038

Table 6.5: DLX Performance for a Compatible Loop Cluster, for Integer Arrays

(a) Speedup of Collective Loop Transformations
due to Register Variables and due to `dlxcc` Optimization†

Phase of	register var, dlxcc		dlxcc -O	
Transformation	instr cnt	cycles	instr cnt	cycles
original code	1 3	1 1	2 0	2 1
instr-scheduled orig			2 0	2 2
unrolled once	1 2	1 4	2 1	2 2
unrolled once, sched		-	2 1	2 3
unrolled 3 times	1 2	1 4	2 0	2 2
unrolled 3x, sched			2 0	2 3
fused original	1 2	1 3	3 1	3 7
fused orig, sched			3 4	3 9
array contracted	1 8	2 0	2 4	2 8
contracted, sched			2 4	2 8
software pipelined	1 8	2 2	2 4	2 8
pipelined, sched			2 1	2 8
pipelined w/ buffer	1 9	2 2	2 4	2 8
pipln w/ buf, sched		-	2 4	2 8

† The speedup for each instruction-scheduled code was computed with respect to the performance of non-optimized code compiled from the same source code

(b) Speedup due to Collective Loop Transformation
over Performance of the Original Code

Phase of	dlxcc		register var, dlxcc		dlxcc -O	
Transformation	instr cnt	cycles	instr cnt	cycles	instr cnt	cycles
instr-scheduled orig					1 0	1 0
unrolled once	1 2	1 2	1 2	1 2	1 3	1 3
unrolled once, sched	–		-		1 3	1 3
unrolled 3 times	1 4	1 3	1 3	1 3	1 4	1 4
unrolled 3x, sched	-	-			1 4	1 4
fused original	1 3	1 2	1 2	1 2	2 2	2 2
fused orig, sched	–		-		2 2	2 3
array contracted	3 6	3 3	5 0	5 0	4 4	4 4
contracted, sched	-				4 4	4 4
software pipelined	3 5	3 2	5 1	5 1	4 3	4 3
pipelined, sched	-				4 3	4 3
pipelined w/ buffer	3 4	3 1	5 0	5 0	4 1	4 1
pipln w/ buf, sched	–	–			4 1	4 1

Table 6.6: Speedup for a Compatible Loop Cluster, using Integer Arrays

results showing the effects of instruction scheduling upon the various transformations. For instance, in the first row of Table 6.5(c) we show the performance of the original code when compiled with "dlxcc -O", without instruction scheduling, and in the second row we show the performance of the same code again when compiled with "dlxcc -O", but this time with the option to use Shieh-Papachristou instruction scheduler, in addition to the other native-compiler optimizations used in the first instance Similar tests were run for each of the remaining codes, as shown in the table

The various tests with instruction scheduling, shown in 6.5(c), show the relative impact of this particular optimization Clearly, scheduling works best upon codes in which there are higher-levels of instruction-level parallelism. For this reason, unrolled codes and fused codes in which there were opportunities for concurrent address and data calculations improved scheduling effectiveness, however, the improvement was comparatively small, i e , less than ten percent, and in no instance was scheduling able to remove all stalls By way of illustration, notice that sufficient stalls remained after the original collection was unrolled once and scheduled, that further unrolling (i.e , unrolling the original collection three times) and scheduling yielded benefit over unrolling once In spite of this benefit, the fused and instruction-scheduled loop performed better, cf a run time of 3712 cycles for "fused orig, sched" as opposed to 5881 cycles for "unrolled 3x, sched" No improvement was made, however, in the array-contracted code and the two software-pipelined codes since the loop body in all three codes was highly flow dependent, providing further indication of a performance constraint caused by exposing too little instruction-level parallelism.

The reason for the degradation in this software-pipelined code, shown in Table 6 5(c), was that the compiler used an unnecessary addi instruction to reload a constant during each iteration of the loop body, rather than retaining this value in register from one iteration to the next And the reason the software-pipelined code which included a buffer statement took even longer to execute was because the DLX architecture was unable to take advantage of the added parallelism provided by the buffering, leaving instead an additional assignment statement (the buffer) to be executed during each iteration of the loop.

The performance of all of the codes might have been improved by filling the branch delay slot with a useful instruction, but this feature had not been implemented yet in version of dlxcc we used All of the NOPs shown in the Table 6 5 were due to this reason Had this transformation been implemented, the relative effect would have been a speedup of less than ten percent. Moreover, the reduction in cycles would likely have been less than the number of NOPs indicated since instruction scheduling likely would not have been able to fill the the full number of branch delay slots that would have been exposed Lastly, notice that the number of NOPs for both sets of the non-optimized codes shown in Tables 6.5(a) and (b) were twice the number as in the optimized set of codes shown in Table 6 5(c) The reason the optimized codes had fewer NOPs is because the optimized code replaced the loop control structure which placed a comparison operation and conditional branch at the top and unconditional jump at the bottom of each loop with a structure that performed the comparison and conditional jump at the bottom of each loop, thereby eliminating the unconditional jump and a NOP.

Tests with Floating-Point Arrays

For this next series of experiments we reran our previous test codes with the arrays in each code first defined as single-precision floating-point, or `float`, rather than `integer` as in the previous tests (see Table 6.7), and then as double precision, or `double`. Constants were not redefined, but remained integer type, therefore, the principle new effect from these experiments, compared with the last, was not the manner of array-element computation necessarily, but rather the effects of register operations, such as data-type conversion and register transfers, upon overall code performance (refer to Figures 6 1 and 6.2)

In certain instances, e g., the non optimized array-contracted code, the single precision codes run almost twenty-percent slower than the corresponding integer codes. The most notable reason for this decrease in performance was the large number of stalls, as can be seen in Table 6 7(a). Also, as suggested in the previous paragraph, contributing to the higher cycle count were the additional instructions required to move integer data values to floating-point registers and then convert these values to floating-point format When scalar values were specified as register type, a thirty to seventy-five-percent improvement in performance occurred, as shown in Table 6 7(b) Notice in Table 6 7(b) that the improvement was reflected in three ways *1)* a reduction in the absolute number of loads and stores, *2)* a commensurate reduction in the number of load stalls, and *3)* a commensurate reduction in the number of instructions. As in the case of the previous integer tests, the use of register variables had no effect on the resulting optimized code, shown in 6 7(c), however the experiments with register variables show, once again, the relative importance of effective register allocation.

(a) Performance of Collective Loop Transformations using **dlxcc** without Optimization

Phase of	integer		floating point		load	FP	instr.	run time
Transformation	loads	stores	loads	stores	stalls	stalls	count	(in cycles)
original code	2707	607	1000	600	1701	400	16551	18652
unrolled once	2157	308	950	600	1851	400	13654	15905
unrolled 3 times	1882	158	925	600	1976	400	12204	14580
fused original	1802	102	900	600	1901	400	13111	15412
array contracted	602	102	900	600	901	400	5111	6412
software pipelined	586	106	1000	692	989	500	5219	6708
pipelined w/ buffer	682	207	1000	690	1086	500	5396	6982

(b) Performance of Collective Loop Transformations using Register Variables, but without **dlxcc** Optimization

Phase of	integer		floating point		load	FP	instr.	run time
Transformation	loads	stores	loads	stores	stalls	stalls	count	(in cycles)
original code	1	1	900	600	501	400	13239	14140
unrolled once	1	1	900	600	501	400	11189	12090
unrolled 3 times	1	1	900	600	501	400	10164	11065
fused original	1	1	900	600	501	400	11209	12110
array contracted	1	1	200	100	201	300	3209	3710
software pipelined	1	1	208	100	204	203	3145	3552
pipelined w/ buffer	1	1	210	100	205	299	3229	3733

(c) Performance of Collective Loop Transformations using **dlxcc -O**

Phase of	integer		floating point		load	FP	instr.	run time
Transformation	loads	stores	loads	stores	stalls	stalls	count	(in cycles)
original code	1	1	900	600	501	400	8455	9356
instr-sched orig	1	1	900	600	300	0	8455	8755
unrolled once	1	1	800	600	501	400	6515	7416
unrolled once, sched	1	1	800	600	150	100	6515	6765
unrolled 3 times	1	1	750	600	501	400	6162	7063
unrolled 3x, sched	1	1	750	600	100	25	6162	6287
fused original	1	1	600	600	401	500	4315	5216
fused orig, sched	1	1	600	600	100	100	4315	4515
array contracted	1	1	200	100	201	300	2611	3112
contracted, sched	1	1	200	100	100	100	2611	2811
software pipelined	1	1	202	100	199	208	2558	2965
pipelined, sched	1	1	202	100	1	112	2558	2671
pipelined w/ buffer	1	1	201	100	197	302	2641	3140
pipln w/ buf, sched	1	1	201	100	0	301	2641	2942

Table 6.7: DLX Performance for a Compatible Loop Cluster, for Single-Precision Floating-Point Arrays

(a) Speedup of Collective Loop Transformations due to Register Variables and due to dlxcc Optimization

Phase of	register var, dlxcc		dlxcc -O	
Transformation	instr cnt	cycles	instr cnt	cycles
original code	1 3	1 3	2 0	2 0
instr-scheduled orig			2 0	2 1
unrolled once	1 2	1 3	2 1	2 1
unrolled once, sched			2 1	2 1
unrolled 3 times	1 2	1 3	2 0	2 1
unrolled 3x, sched		-	2 0	2 3
fused original	1 2	1 3	3 0	3 0
fused orig, sched			3 0	3 1
array contracted	1 6	1 7	2 0	2 1
contracted, sched			2 0	2 3
software pipelined	1 7	1 9	2 0	2 3
pipelined, sched			2 0	2 5
pipelined w/ buffer	1 7	1 9	2 0	2 2
pipeln w/ buf, sched			2 0	2 4

† The speedup for each instruction-scheduled code was computed with respect to the performance of non-optimized code compiled from the same source code

(b) Speedup due to Collective Loop Transformations over Performance of the Original Code

Phase of	dlxcc		register var, dlxcc		dlxcc -O	
Transformation	instr cnt	cycles	instr cnt	cycles	instr cnt	cycles
instr-scheduled orig					1 0	1 1
unrolled once	1 2	1 2	1 2	1 2	1 3	1 3
unrolled once, sched					1 3	1 4
unrolled 3 times	1 4	1 3	1 3	1 3	1 4	1 3
unrolled 3x, sched	-				1 4	1 5
fused original	1 3	1 2	1 2	1 2	2 0	1 8
fused orig, sched					2 0	2 1
array contracted	3 2	2 9	4 1	3 8	3 2	3 0
contracted, sched					3 2	3 3
software pipelined	3 2	2 8	4 2	4 0	3 3	3 2
pipelined, sched					3 3	3 5
pipelined w/ buffer	3 1	2 7	4 1	3 8	3 2	3 0
pipln w/ buf, sched					3 2	3 2

Table 6.8: Speedup for a Compatible Loop Cluster using Single-Precision Floating Point Arrays

A second factor which is evident from Table 6.7(b) is the impact of array contraction. As in the previous integer tests, array contraction had the most impact of any of the loop transformations employed (recognizing, however, that contraction would not have been possible unless loop fusion were first performed). In Table 6.7(b), for example, there was a reduction in the number of floating-point loads from 900, for the "fused original" code, to just 200, for the "array contracted" code, a reduction in the number of stores from 600 to 100, a reduction in the total number of stalls from 901 to 501, and a reduction in the number of instructions from 11209 to 3209. The overall effect was a speedup of 3.3 for the array-contracted code over the performance of the code in which the loops were fused.

The statistics in Table 6.7(b) also point out a shortcoming in common subexpression elimination. Notice the large difference between the numbers of loads and stalls for the "fused original" code compared with the numbers for the "array contracted" code. This difference was due to the fact that the array references tend to loose their identity during intermediate-code generation, as part of processing by the compiler back end. Specifically, during processing, each array reference is translated into a series of addressing instructions followed by a load or a store, thereby making it difficult to associate a particular register with a particular array element when the array element is later referenced. As a consequence, each element referenced is stored after its data value is assigned and then naively reloaded from memory, each time the element is later referenced rather than using the element's live register value, as would be the case with scalar variables. On the other hand, array contraction makes the association of array elements and register explicit by replacing complex reference sequences with simpler scalar references. As a result, effect is essentially the same as would be accomplished by common sub-expression elimination, were common subexpression elimination actually able to handle such cases.

Another significant observation regarding the statistics, this time in Table 6.7(c), is with respect to the effectiveness of the combination of instruction scheduling and loop fusion as compared with the effectiveness of the combination of instruction scheduling and loop unrolling, in particular, "unrolled 3x, sched". As shown in the table, just as many load stalls were eliminated in either case, and although the unrolled code resulted in fewer stalls after instruction scheduling than did the fused code, the fused code required 1817 fewer instructions, allowing it to execute in thirty-percent less time.[9]

[9] Later in this section we provide evidence to suggest why this difference might sometimes be even more

The results of the experiments with transformations in which arrays were defined as double-precision floating-point, i.e., as "`double`", are predictably similar to the results in which arrays were defined as "`float`" (see Table 6.9). The reason, of course, is the that the translated code was essentially the same, except for the load, store, and floating-point conversion instructions, for which corresponding double precision instructions were used. Consequently, the optimized instruction scheduled software pipelined code, shown in Table 6.9(c), achieved the highest overall performance, as in the case of both integer-defined arrays and single precision floating point arrays, requiring only 2813 cycles for a speedup of 3.4 over the performance of the optimized code for the original loop collection

Lastly, one more minor point with regard to the floating-point test results. Notice that in Tables 6.7(b) and (c) (page 6.7), and also in Tables 6.9(b) and (c), that 200 or so floating-point loads occurred for the array-contracted code and software pipelined codes; whereas in the corresponding integer experiment there were only 100 or so, cf Table 6.5 (page 122). The additional hundred loads, in this case, were due to the fact that the compiler converted the constant 99 from integer to float and stored the value with other global constants, to avoid another run-time data conversion, whereas, in the case of the integer codes the constant 99 was merely an immediate operand of the `add` instruction used to compute the value of the scalar variable `d` (refer to Table 6.5).

Cache Simulation Results

The simulations we have described so far focused on only one aspect of performance (albeit an important one), and that was processor performance. However, an equally important aspect with respect to the performance of collective loop transformations is memory performance. To better understand the effects of memory references upon collective loop transformation, we used the *dinero* cache simulator [HS84], along with address traces generated from each of the optimized codes shown in Table 6.5(c) (page 122), Table 6.7(c) (page 126) and Table 6.9(c) (page 130). The results of these simulations are shown in Table 6.10.

The data in Table 6.10 is organized according to four major classifications: *1)* total number of instruction-cache references and total number of misses, *2)* total number of data-cache references and total number of misses, *3)* combined instruction-cache and data-cache results, and *4)* total number of memory fetches, which in this case also corresponds to the number of 32-bit words fetched from memory. As noted at the

(a) Performance of Collective Loop Transformations using `dlxcc` without Optimization

Phase of Transformation	integer loads	integer stores	floating point loads	floating point stores	load stalls	FP stalls	instr. count	run time (in cycles)
original code	2708	608	1000	600	1700	400	16553	18653
unrolled once	2158	308	950	600	1850	400	13653	15903
unrolled 3 times	1883	158	925	600	1975	400	12203	14578
fused original	1803	103	900	600	1900	400	13113	15413
array contracted	603	103	900	600	900	400	5113	6413
software pipelined	595	116	1000	692	980	500	5239	6719
pipelined w/ buffer	793	319	1000	690	975	500	5620	7095

(b) Performance of Collective Loop Transformations using Register Variables, but without `dlxcc` Optimization

Phase of Transformation	integer loads	integer stores	floating point loads	floating point stores	load stalls	FP stalls	instr. count	run time (in cycles)
original code	2	2	900	600	500	400	13241	14141
unrolled once	2	2	900	600	500	400	11191	12091
unrolled 3 times	2	2	900	601	500	400	10167	11067
fused original	2	2	900	600	500	400	11211	12111
array contracted	2	2	200	100	200	300	3211	3711
software pipelined	2	2	208	100	203	203	3147	3553
pipelined w/ buffer	2	2	210	100	204	299	3231	3734

(c) Performance of Collective Loop Transformations using `dlxcc -O`

Phase of Transformation	integer loads	integer stores	floating point loads	floating point stores	load stalls	FP stalls	instr. count	run time (in cycles)
original code	2	2	900	600	500	400	8457	9357
instr-scheduled orig	2	2	900	600	300	0	8457	8757
unrolled once	2	2	800	600	500	400	6517	7417
unrolled once, sched	2	2	800	600	150	100	6517	6767
unrolled 3 times	2	2	750	600	500	400	6164	7064
unrolled 3x, sched	2	2	750	600	100	25	6164	6289
fused original	2	2	600	600	400	500	4317	5217
fused orig, sched	2	2	600	600	100	100	4317	4517
array contracted	2	2	200	100	200	300	2613	3113
contracted, sched	2	2	200	100	100	100	2613	2813
software pipelined	2	2	202	100	198	208	2560	2966
pipelined, sched	2	2	202	100	1	112	2560	2673
pipelined w/ buffer	2	2	201	100	196	302	2643	3141
pipeln w/ buf, sched	2	2	201	100	0	301	2643	2944

Table 6.9: DLX Performance for a Compatible Loop Cluster, for Double-Precision Floating-Point Arrays

beginning of Section 6.2, we assumed an 8K-byte instruction cache and a 64K-byte data cache, both being 1-way set associative with 128-byte lines and using a least recently-used (LRU) line-replacement policy. These particular configurations were used because they are representative of what is commercially available, for example, these they are used on the IBM RISC System/6000.

There are three results from this particular set of simulations which are interesting. The first is with respect to cache performance on the unrolled codes, as opposed to the fused codes, the second is with respect to the reduction in data cache misses as a result of array contraction, and the third is with respect to both instruction and data cache misses for the array-contracted codes, compared with the software-pipelined codes.

First, notice that in all three cases, shown in Tables 6.10(a)–(c) (i.e., for codes for integer-defined arrays, for single-precision floating-point arrays and for double precision floating-point arrays), the unrolled codes had a higher number of misses than the corresponding fused codes, and the number of misses increased as the amount of unrolling increased. This effect is readily explained by the fact that each loop in the code for the original loop collection, and each loop of the unrolled codes, brought in new instructions, with each loop body that was executed. Similarly, as the loop bodies increased in size because of unrolling, additional cache lines were likewise needed, further increasing the number of instruction-cache misses. On the other hand, by consolidating the loops through loop fusion, the fused loop body (although larger perhaps than any of the separate loops) only introduced code once, thereby achieving better cache-line utilization than the codes without loop fusion.

The second effect that is observable from the cache-simulation results shown in Table 6.10 is with respect to the reduction in data-cache misses due to array contraction. Notice in each table that the number of data cache misses was the same for test codes before array contraction and the same for codes afterwards, although miss ratios vary from one code to the next because of the decreasing number of memory references for each successive code. The difference, however, in ratios is less than half a percent. Nevertheless, a four- to five fold reduction in cache misses occurred as a result of contraction, and as with the finding in the preceding paragraph, the effect across arrays of different data type was uniform.

Now, compare the impact of misses upon the overall performance of the respective unrolled codes and fused codes, considering both processor performance and cache performance. If we assume that the first word referenced within a cache line not currently in cache requires 8 cycles to fetch, and each remaining word to complete

(a) Performance of Loop Transformations, for Integer Arrays

Phase of	instruction cache		data cache		total		memory
Transformation	refn.	misses	refn.	misses	refn.	misses	fetches
original code	7950	5	1400	20	9350	25	800
instr-scheduled orig	7951	5	1401	20	9352	25	800
unrolled once	6160	7	1350	20	7510	27	864
unrolled once, sched	6161	7	1351	20	7512	27	864
unrolled 3 times	5731	10	1325	20	7059	30	960
unrolled 3x, sched	5735	10	1326	20	7061	30	960
fused original	3612	3	1100	20	4712	23	736
fused orig, sched	3613	3	1101	20	4714	23	736
array contracted	1809	2	200	4	2009	6	192
contracted, sched	1810	2	201	4	2011	6	192
software pipelined	1850	3	197	4	2047	7	224
pipelined w/ buffer	1941	3	196	4	2127	7	224

(b) Performance, for Single-Precision Floating-Point Arrays

Phase of	instruction cache		data cache		total		memory
Transformation	refn.	misses	refn.	misses	refn.	misses	fetches
original code	8456	6	1502	21	9958	27	864
instr-scheduled orig	8456	6	1502	21	9958	27	864
unrolled once	6516	7	1402	21	7918	28	896
unrolled once, sched	6516	7	1402	21	7918	28	896
unrolled 3 times	6163	11	1352	21	7515	32	1024
unrolled 3x, sched	6163	11	1352	21	7515	32	1024
fused original	4316	3	1202	21	5518	24	768
fused orig, sched	4316	3	1202	21	5518	24	768
array contracted	2616	2	302	5	2914	7	224
contracted, sched	2612	2	302	5	2914	7	224
software pipelined	2559	4	304	5	2863	9	288
pipelined w/ buffer	2642	4	303	5	2945	9	288

(c) Performance, for Double-Precision Floating-Point Arrays

Phase of	instruction cache		data cache		total		memory
Transformation	refn.	misses	refn.	misses	refn.	misses	fetches
original code	8458	6	3004	40	11462	46	1472
instr-scheduled orig	8458	6	3004	40	11462	46	1472
unrolled once	6518	7	2804	40	9322	47	1504
unrolled once, sched	6518	7	2804	40	9322	47	1504
unrolled 3 times	6165	11	2704	40	8869	51	1632
unrolled 3x, sched	6165	11	2704	40	8869	51	1632
fused original	4318	3	2404	40	6722	43	1376
fused orig, sched	4318	3	2404	40	6722	43	1376
array contracted	2614	2	604	9	3218	11	352
contracted, sched	2614	2	604	9	3218	11	352
software pipelined	2561	4	608	15	3169	19	608
pipelined, sched	2561	4	608	15	3169	19	608
pipelined w/ buffer	2644	5	606	15	3250	20	640
pipeln w/ buf, sched	2644	5	606	15	3250	20	640

Table 6.10. Dinero Cache Performance for a Compatible Loop Cluster

the line is loaded in one cycle (assuming references that occur immediately occur in increasing order), the incremental effect due to data-cache misses alone would be $(8-1)\cdot 20 = 140$ cycles to execute the non-fused integer and single-precision floating point codes, and a 280 cycles to execute the double-precision codes.[10] Considering both instruction-cache and data-cache misses for the double precision codes, the increase would be as much as 364 cycles, for "unrolled 3x, sched", corresponding to 52 cache misses altogether; see Table 6.10(c). And, compared with the overall performance of the double-precision instruction-scheduled fused code—including cache miss penalties—the "unrolled 3x, sched" code is about thirty percent slower.

Although the above two findings clearly indicate that the fused code in the experiments will significantly out-perform any of the unrolled codes, the actual machine timings, shown in Table 6.1 (page 106), support neither result.

The last effect mentioned above concerns the number of instruction- and data-cache misses for the array-contracted codes, as opposed to the corresponding numbers for the two software-pipelined codes. Notice that in all instances, the number of instruction-cache misses and the number of data-cache misses for the software-pipelined codes were at least as high, and in most instances, higher than they were for the corresponding array-contracted codes. This effect was caused by the prolog and epilog of these codes being in different cache lines than the corresponding loop bodies. Since the prolog and epilog of software-pipelined code represents a relatively small number of instructions, however, the number of cache lines affected is small, as well as the overall effect on program performance. Consequently, this effect too is not as apparent from the actual timings in Table 6.1, as it is from the cache simulation results shown in Table 6.10.

6.2.2 Tests with a Non-Compatible Cluster of Loops

The same processor and cache configurations were used to evaluate the performance of various transformations involving both naive (sub-optimal) and efficient (optimal) transformation of the non-compatible loops, shown in Figure 6.3, as were used for the compatible-loop transformations described in the last section, Section 6.2.1. As in the last section, our objectives with respect to these particular experiments were essentially the same as they were for the corresponding timing experiments with

[10] Eight cycles plus one cycle for each remaining word in the cache line is what is required on the RS/6000.

partitioned clusters, described in Section 6.1.2. Specifically, we wanted to again *1)* measure the effect of optimal loop partitioning over naive sub-optimal partitioning, *2)* isolate the effects of loop fusion and array contraction when applied at various stages of collective transformation, *3)* determine the impact of instruction scheduling, and *4)* compare the simulated performance of the various loop transformations with the performance of the actual machines, described in Table 6.3

Processor Simulation Results

The results of the DLX processor simulations in which we used a non-compatible loop cluster are shown in Table 6.11. For each of these experiments the codes were optimized, i e , compiled using "`dlxcc -O`". In Table 6.11(a) we show the results of executing the codes in Figure 6 3 with arrays defined as `integer`, in Table 6 11(b) the results with arrays defined as `float`, and Table 6 11(c) the results with the arrays defined as `double` To make the performance impact easier to see, we show the speedups of the respective transformations, over the performance of the optimized original loop cluster, in Table 6.12.

As in the case of our experiments with a compatible loop collection, the codes with single-precision floating-point arrays performed identically to the codes with double-precision arrays, cf. Tables 6.11(b) and (c). Again, the reason the results from these two sets of codes were same was the one-to-one correspondence between single-precision and double-precision instructions for loads, stores, and data conversion Differences in performance between the two sets of codes exist, however, these differences relate to cache-memory effects, not processor performance, as we show in the next section.

In general, the sub-optimal code, "contr naive, sched", resulted in a thirty- to forty-percent speedup in performance over the original non-compatible loop cluster, whereas, the optimally partitioned code, "contr eff, sched", achieved a seventy- to ninety-percent speedup Moreover, the speedups for the sub-optimal code were from twenty to thirty-percent higher on the simulator than they were on the actual machines, cf Table 6.4 (page 116) and Table 6.12. The results for the optimally partitioned code, although closer, were less consistent. For example, the integer and single-precision floating-point codes achieved higher speedups on the simulator, but double precision floating-point codes achieved higher speedups on the actual machines, including the vectorized double-precision codes which were run on the 3090 This effect is partially explained in the case of the RS/6000 by the way single-precision

(a) Performance of Loop Transformations, for Integer Arrays

Phase of	integer		load	FP	instr	run time
Transformation	loads	stores	stalls	stalls	count	(in cycles)
original code	1000	800	600	800	10050	11150
instr-scheduled orig	1000	800	0	800	10050	10850
contracted original	900	800	400	800	9950	11150
contr original, sched	900	800	0	800	9950	10750
fused naive	800	800	600	800	8326	9726
fused naive, sched	800	800	0	800	8326	9126
fused original	600	700	300	800	7328	8128
fused orig, sched	600	700	0	800	7328	8128
fused efficient	800	800	600	100	7225	8225
fused eff, sched	800	800	0	100	7225	7625
contr efficient	400	300	200	100	5721	6321
contr eff, sched	400	300	0	400	5721	6121

(b) Performance, for Single-Precision Floating-Point Arrays

Phase of	floating point		load	FP	instr	run time
Transformation	loads	stores	stalls	stalls	count	(in cycles)
original code	1800	800	1200	800	11951	13951
instr-scheduled orig	1800	800	700	700	11951	13351
contracted original	1700	800	1200	800	11851	13851
contr original, sched	1700	800	400	1200	11851	13451
fused naive	1500	800	1200	800	10025	12025
fused naive, sched	1500	800	500	600	10025	11125
fused original	1400	700	1200	600	9526	11326
fused orig, sched	1400	700	200	900	9526	10626
fused efficient	1600	800	1100	700	9525	11325
fused eff, sched	1600	800	300	800	9525	10625
contr efficient	1200	300	900	700	8021	9621
contr eff, sched	1200	300	0	800	8021	8821

(c) Performance, for Double-Precision Floating-Point Arrays

Phase of	floating point		load	FP	instr	run time
Transformation	loads	stores	stalls	stalls	count	(in cycles)
original code	1800	800	1200	800	11951	13951
instr-scheduled orig	1800	800	700	700	11951	13351
contracted original	1700	800	1200	800	11851	13851
contr original, sched	1700	800	400	1200	11851	13451
fused naive	1500	800	1200	800	10025	12025
fused naive, sched	1500	800	500	600	10025	11125
contracted naive	1400	700	1200	600	9526	11326
contr naive, sched	1400	700	200	900	9526	10626
fused efficient	1600	800	1100	700	9525	11325
fused eff, sched	1600	800	300	800	9525	10625
contr, efficient	1200	300	900	700	8021	9621
contr eff, sched	1200	300	0	800	8021	8821

Table 6.11: DLX Performance for a Non-Compatible Loop Cluster

Table 6.12. Speedup due to Collective Loop Transformation for a Non-Compatible Loop Cluster, using `dlxcc -O`

Phase of Transformation	Integer		Single-Precision		Double-Precision	
	instr cnt	cycles	instr cnt	cycles	instr cnt	cycles
instr-scheduled orig	1.0	1.1	1.0	1.0	1.0	1.0
contracted original	1.0	1.0	1.0	1.0	1.0	1.0
contr original, sched	1.0	1.1	1.0	1.0	1.0	1.0
fused naive	1.2	1.2	1.2	1.2	1.2	1.2
fused naive, sched	1.2	1.3	1.2	1.3	1.2	1.3
contracted naive	1.4	1.4	1.3	1.2	1.3	1.2
contr naive, sched	1.4	1.4	1.3	1.3	1.3	1.3
fused efficient	1.4	1.4	1.3	1.2	1.3	1.2
fused eff, sched	1.4	1.5	1.3	1.3	1.3	1.3
contr efficient	1.8	1.8	1.5	1.5	1.5	1.5
contr eff, sched	1.8	1.9	1.7	1.6	1.5	1.6

floating-point codes are handled, i.e., by performing all floating-point calculations in double precision; however, this explanation is insufficient, in general In the the next section, where we describe simulated cache effects, we examine this anomaly further.

Cache Simulation Results

As pointed out previously, processor performance alone does not tell the whole story with respect to the performance of naive versus efficient cluster partitioning. Therefore, in this section we look at cache-memory effects, using test results from the dinero cache simulator, as we did before. For these experiments, we used trace files generated during DLX simulation of the codes evaluated in the preceding section, and the cache configuration used for experiments with a compatible loop cluster, described in Section 6.2.1: an 8K-byte instruction cache and a 64K-byte data cache, both being 1-way set associative with 128-byte lines. The results of these experiments are shown in Table 6.13

The results in Tables 6.13(a)-(c) are what we would expect to see: No arrays were actually eliminated by array-contraction of the original code, cf. Figures 6.3(a) and (b) (page 113), so cache performance was the same for the respective two codes, i.e., the "original code" and the "contracted original". Loop fusion of the optimally partitioned code resulted in the same number of array references as loop fusion of the sub-optimally partitioned code, cf. Figures 6.3(c) and (e); consequently, cache

(a) Performance of Loop Transformations, for Integer Arrays

Phase of	instruction cache		data cache		total		memory
Transformation	refn.	misses	refn.	misses	refn	misses	fetches
original code	10051	6	1800	26	11851	32	1024
instr-scheduled orig	10051	6	1800	26	11851	32	1024
contracted original	9951	6	1700	26	11651	32	1024
contr original, sched	9951	6	1700	26	11651	32	1024
fused naive	8327	4	1600	26	9927	30	960
fused naive, sched	8327	4	1600	26	9927	30	960
contracted naive	7329	4	1300	23	8629	27	864
contr naive, sched	7329	4	1300	23	8629	27	864
fused efficient	7226	4	1600	26	8826	30	960
fused eff, sched	7226	4	1600	26	8826	30	960
contr efficient	5722	3	700	10	6422	13	416
contr eff, sched	5722	3	700	10	6422	13	416

(b) Performance, for Single-Precision Floating-Point Arrays

Phase of	instruction cache		data cache		total		memory
Transformation	refn	misses	refn.	misses	refn	misses	fetches
original code	11952	6	2600	27	14552	33	1056
instr-scheduled orig	11952	6	2600	27	14552	33	1056
contracted original	11852	6	2500	27	14352	33	1056
contr original, sched	11852	6	2500	27	14352	33	1056
fused naive	10026	5	2300	27	12326	32	1024
fused naive, sched	10026	5	2300	27	12326	32	1024
contracted naive	9527	5	2100	24	11627	29	928
contr naive, sched	9527	5	2100	24	11627	29	928
fused efficient	9526	5	2400	27	11926	32	1024
fused eff, sched	9526	5	2400	27	11926	32	1024
contr efficient	8022	4	1500	11	9522	15	480
contr eff, sched	8022	4	1500	11	9522	15	480

(c) Performance, for Double-Precision Floating-Point Arrays

Phase of	instruction cache		data cache		total		memory
Transformation	refn	misses	refn.	misses	refn	misses	fetches
original code	11952	7	5200	57	17152	64	2048
instr-scheduled orig	11952	7	5200	57	17152	64	2048
contracted original	11852	7	5000	57	16852	64	2048
contr original, sched	11852	7	5000	57	16852	64	2048
fused naive	10026	5	4600	57	14626	62	1984
fused naive, sched	10026	5	4600	57	14626	62	1984
contracted naive	9527	5	4200	51	13727	56	1792
contr naive, sched	9527	5	4200	51	13727	56	1792
fused efficient	9526	5	4800	57	14326	62	1984
fused eff, sched	9526	5	4800	57	14326	62	1984
contr efficient	8022	4	3000	25	11022	29	928
contr eff, sched	8022	4	3000	25	11022	29	928

Table 6.13: Dinero Cache Performance for a Non-Compatible Loop Cluster

performance of these two codes, i e., "fused naive" and "fused efficient", again was the same. And, the number of arrays remaining after array contraction of the optimally partitioned code was much fewer than the number remaining after contraction of the sub-optimally partitioned code, cf 6 3(d) and (f), consequently, the number of misses in this case was much lower, both for the instruction cache (because of the fewer number of indexing instructions) and for the data cache (because of the fewer number of loads and stores) (see "contr naive sched" and "contr eff. sched").

What is surprising from these results is that they fail to explain the differences in performance noted in the previous section First, since the processor performance of single-precision and double-precision floating-point computations were the same, cf Tables 6 11(b) and (c), and twice as many data-cache misses occurred with the double-precision codes as with the single-precision codes, cf. Table 6.13(b) and (c), one would expect the timing results for double-precision floating-point codes to be much less than the corresponding single-precision codes, but this was not the case, as was pointed out in the previous section. Also, since the "fused naive" code and the "fused efficient" code both required fewer processor cycles than the corresponding "original code", cf Table 6 11(a) (c), and both had a lower number of cache misses, one would expect the timing results for the fused codes to be commensurately less than the results for the corresponding original code; however, this too was not the case, again as pointed out. Unfortunately, for neither of these two anomalies are we able to give a satisfactory explanation, aside from perhaps timing errors and/or rounding errors in the regression calculation used to compute eps.

6.3 Summary Observations

The experiments reported in this chapter show that the benefit from collective transformation can be substantial, sometimes resulting in more than a four-fold speedup in performance. The benefit derives not from any single transformation, per se, but from a combination of transformations, with array contraction having the most pronounced effect Besides providing an indication of the overall benefit of collective transformation, our experiments support several commonly held perceptions regarding the effectiveness of the individual transformations employed, such as loop unrolling, loop fusion, array contraction, and software pipelining. To tie these results together, we now summarize our findings, both with respect to the above transformations and with respect to the related transformations used for comparison.

For non-vectorized codes, loop unrolling decreased instruction counts by reducing loop control overhead, such as compares, jumps, and delayed branch NOPs. Unrolling also improved the effectiveness of instruction scheduling by increasing loop body size. On the other hand, it also increased instruction cache misses and, hence, memory fetches, offsetting some of the previous positive effects. The overall effect, however, was a ten- to twenty-percent improvement in performance for the test codes run on the Sun and slightly higher performance improvement for codes run on the RS/6000. This was because the superscalar RS/6000 was better able to take advantage of the instruction-level parallelism provided by the transformation. For the 3090, however, the effect was opposite, with a ten- to twenty-percent decrease in performance. This was because unrolling combines instructions within a loop body in a way that impedes efficient vectorization increasing the stride and number of vector instructions in the transformed code.

As expected, instruction scheduling improved processor performance by reducing load stalls and floating-point stalls. Although these reductions were significant, scheduling did not eliminate stalls entirely in most instances. The effectiveness of instruction scheduling was improved by both loop unrolling and loop fusion, but the effect of scheduling on array contracted-codes was more apt to be neutral, i.e., sometimes benefit was obtained, but, not always. This latter observation is explained by the fact that contracted code is strongly flow dependent, leaving little latitude with respect to the order in which instructions might appear. Instruction scheduling likewise had no impact on memory fetching. This might be a consideration when compared to software pipelining (as we explain below).

Loop fusion also improved code performance, by reducing loop control overhead and by creating a larger loop body upon which additional transformations, such as array contraction and instruction scheduling, could be performed. It also slightly reduced misses in the instruction cache, decreasing the number of memory fetches. The overall improvement from fusion was moderately low, however often less than ten percent. Moreover, in certain instances, the performance of fused code in combination with other optimizations was worse than without it. An instance in which lower performance occurred was the case of floating point codes on the RS/6000. A second instance was the case of vectorizable codes on the 3090. In spite of its limitations loop fusion served an important purpose by facilitating array contraction.

Array contraction was the single most effective transformation applied. The benefit of this transformation was attributable to a decrease in the number of array indexing operations which accompany this transformation and a corresponding decrease in memory references, primarily within the data cache. The only instance in which array contraction resulted in lower performance was in the case of vectorization of loops for which there were a high percentage of double-precision floating-point array references. In spite of its performance benefit over the other transformations tested, none of the commercial compilers used in these experiments actually included array contraction as part of its standard repertoire of optimizing transformations.

In general, software pipelining, loop fusion, and loop unrolling were found to be inappropriate for a vector architecture since each of these transformations usually impeded vectorization. For this reason, our following remarks pertain primarily to the scalar and superscalar architectures used during the experiments, rather than to the vector architecture.

As an instruction scheduling technique, software pipelining was less effective than the instruction schedulers provided by the native compilers. Although it did improve code performance in most instances, eliminating stalls by distancing long-latency operations, it seldom did the job as well as as the native instruction scheduler. This was because the transformation itself is insensitive to the particular sequences of instructions which cause hardware stalls. However, when software pipelining was used in conjunction with the native instruction scheduler, improvement was observed in instances involving a mix of integer and floating-point operations, for both scalar and superscalar machines. This suggests that the primary purpose for software pipelining should be to increase instruction-level parallelism in machines capable of higher levels of concurrency. Furthermore, in these situations, software pipelining which used a buffering instruction generally degraded performance over plain software pipelining. For this reason, we believe buffering without special hardware support should not be included as part of the software-pipelining transformation.

Our experiments suggest that loop fusion should be performed prior to loop unrolling. This is because fusion is less apt to incur the instruction-cache miss penalty that sometimes accompanies unrolling, yet the effect of either transformation upon instruction scheduling is the same: they both increase the number of available instructions for scheduling by increasing loop-body size. In this respect, loop fusion might best be performed by a compiler preprocessor. Likewise, array contraction might also be performed by the preprocessor, since all of the information required to perform contraction is readily available at the time of preprocessing, and the impact upon later code-improving transformations, in all cases, is positive.

Our experiments also suggest that loop unrolling should not be done automatically by the compiler. Rather, it should used only to the extent necessary for effective instruction scheduling. With this observation in mind, we envision three potentially effective approaches to implementing this particular transformation: *1)* to unroll loops at a high level (perhaps within the preprocessor), based upon the use of compiler directives, and use profile information to determine whether unrolling is beneficial, *2)* to unroll loops at the level of the Abstract Syntax Tree (AST), based upon fixed global criteria, such as numbers and types of instructions, or *3)* to collect stall information during an initial scheduling pass, then unroll and reschedule instructions in those instances in which a specified stall threshold is exceeded.

With regard to loop cluster partitioning, our experiments show that it is of greatest importance to minimize the number of intermediate arrays referenced between code blocks, since the difference between optimally partitioned code and sub-optimally partitioned code can be substantial, with optimally partitioned code running almost twice as fast in some instances. The disparity in performance is due to the fact that most of the benefit from loop transformation derives, not from fusion, but from array contraction. Again, the primary effect of loop fusion is to increase the amount of contraction possible, whereas, contraction eliminates costly array indexing operations and memory operations, which constitute the largest share of instructions executed in situations of this type. As a result, if partitioning analysis fails to result in a transformation which employs the minimum number of arrays, the performance penalty can be severe.[11]

[11] We provide just such an example of how this can occur in Chapter 4.

Chapter 7

Conclusions

In this final chapter we briefly summarize the most significant findings from our research and identify suitable topics for further investigation. The purpose of this research, once again, was to assess the feasibility of using collective loop analysis in optimizing compilers for advanced uniprocessor architectures and to determine the benefit derivable when appropriate transformations are applied to compatible loop clusters, as a result of collective analysis. It is within this context that our contributions belong

7.1 Achievements

The following are what we consider to be our major contributions with respect to the study of collective analysis and transformation of loop clusters:

1. Recognition of the fact that the Sarkar-Gao collective analysis method could not be used to transform non-compatible loop clusters for codes written for uniprocessors Although the Gao-Sarkar method identifies least-cost non-compatible edges which can be applied to transformation of multiprocessor codes, this information is insufficient to partition code into fusionable clusters. (See Chapter 4, Section 4.1, page 58.)

2. Recognition of the fact that forks in the Loop Communication Graph (LCG) frequently represent the least-cost arcs to cut when partitioning a an LCG. This observation is based upon the fact that the output edges at a fork often

represent the transfer of a single array, whereas any other cuts which would be necessary to break an odd-length cycle in the interference graph (IG) necessarily affect a minimum of two arrays. Therefore no cut of an isolated cycle can be less costly than a cut at the fork. (See Chapter 3, Section, 3.1, 31.)

3. Development of an efficient coloring technique based upon a breadth first traversal of the of the IG, starting from the nodes which correspond to the output code blocks of the loop cluster (Algorithm II, page 18)

4. Development of an effective loop partitioning heuristic which builds upon the previous algorithm and which achieves optimal transformation, in many instances (see Algorithm III, page 45)

5. Recognition of the fact that the collective loop fusion algorithm, proposed by Gao, Olsen, Sarkar, and Thekkath, is non-optimal in situations having adjacent odd-length cycles in the IG. The reason collective loop fusion fails to achieve optimality in these instances is that the algorithm considers non compatible arcs to be constant, whereas the compatibility status of any particular arc is determined by the order in which odd-length cycles in the IG are cut. (See Chapter 4, Section 4.2.3, page 71.)

6. Experimental results to show the extent of speedup achievable from high level collective loop transformations, such as loop fusion and array contraction, and low-level transformations, such as software pipelining, for several modern high performance computer architectures, such as a scalar RISC workstation, a superscalar RISC workstation, and a mainframe computer with attached vector processing facility. (See Chapter 6, Section 6.1, page 101.)

7. Experimental results to show the effects of collective transformations, such as those mentioned above, on codes run on a RISC processor simulator and a cache simulator. For these tests, comparison was made with respect to the stages of successive transformation, as well as with respect to alternative transformations, such as loop unrolling. (See Chapter 6, Section 6.2, page 118.)

8. Lastly, experimental to show the relative effect of various cluster partitioning strategies, from sub-optimal naive partitioning to optimal partitioning. (See Chapter 6, Section 6.1.2, page 112, and Section 6.2.2, page 133.)

7.2 Topics for Further Research

Based upon the preceding progress, we suggest the following topics for follow-on research:

1. Determining an optimal algorithm to solve the collective loop fusion problem for non-compatible loops or establishing a proof that the problem is NP-Complete.

2. Implementing the algorithms for collective loop analysis in a real compiler.

3 Benchmarking transformation performance on multiprocessor systems and comparing the results to those described for the various uniprocessors tested, and lastly

4 Determining the incidence of eligible loop clusters in codes taken from actual practice and determining the extent of benefit realizable as a result of transformation in these instances

The second item above would involve implementing the relevant transformations required to isolate eligible clusters of loops and to make the loops within these clusters conformable for analysis (transformations described in Chapter 4), as well as implementing the loop analysis and partitioning algorithms described in Chapters 2 and 3

Appendix A

Sample Code Listings

The following listings are extracts from the codes used to measure the performance of the various loop transformations discussed in Chapter 6. They are included merely to illustrate how the actual timings were taken. For tests ran on the Sun SPARC 1/490 and the IBM RISC System/6000, four programs were used altogether. The first, called **mksrc**, is the shell script used as the driver to take timings of the various versions of compatible loops (see Figures 6.1 and 6.2). The second, called `comptst.F`, contains the source code itself, showing implementation of the timing loop. The third, which is not shown, is a modified version of the first, used as a driver for the non compatible loop tests, and the fourth, called `partst.F`, shows the timing loop and test codes for the non-compatible loop transformations (see Figure 6.3). For experiments ran on the IBM 3090/VF, each test code was ran separately. A sample of one of the codes, called **MOREFF**, is shown after the `partst.F` listing. The final listing is an aggregation of the preceding test codes translated into C for use with the DLX processor simulator and Dinero cache simulator. All codes used during the experiments reported in this thesis were tested prior to the respective timing test, to verify loop transformation correctness.

```
# execute the test codes in "src" and place the assembly-language listing,
# along with the test results in .src file.  R. Olsen,
# [illegible] Jan [illegible] 12 24:16 EST 1992

base=50  # loop lower bound
nmax=1000 # loop upper bound
incr=50  # loop increment
iter=10  # number of iterations of the test body

ldefs=" -DBASE=${base} -DNMAX=${nmax} -DINCR=${incr} -DITER=${iter} "
src=loopoptst.F

if test -f ${src}
then
   echo -n # ensure the test source file is present
else
   echo "${src} not found"
   exit
fi

for opt in opt nopt
do
   if test ${opt} = nopt
   then
      echo "running unoptimized versions"
      cflags="-U"
   else
      echo "running optimized versions"
      cflags="-O -U"
   fi
   for type in int float double
   do
      case ${type} in
         int) tdef="-DTYPE=INTEGER";;
         float) tdef="-DTYPE=REAL*4";;
         double) tdef="-DTYPE=REAL*8";;
      esac
      for prog in orig unroll1 unroll3 fused contr pipe pipebuf
      do
         case ${prog} in
            orig) tst="-DORIG_TST";;
            unroll1) tst="-DUNROLL1_TST";;
            unroll3) tst="-DUNROLL3_TST";;
            fused) tst="-DFUSED_TST";;
            contr) tst="-DCONTR_TST";;
            pipe) tst="-DPIPE_TST";;
            pipebuf) tst="-DPIPEBUF_TST";;
         esac
         f77 ${cflags} ${tst} ${tdef} ${ldefs} -o ${prog}.${type}.${opt} ${src}
         f77 -S ${cflags} ${tst} ${tdef} ${ldefs} -o test.s ${src}
         if test -f test.s
         then
            echo "cat test.s > ${prog}.${type}.${opt}.src"
            cat test.s > ${prog}.${type}.${opt}.src
            if test -f ${prog}.${type}.${opt}
            then
               echo "executing ${prog}.${type}.${opt}"
               ${prog}.${type}.${opt} >> ${prog}.${type}.${opt}.src
            else
               echo "==> file \"${prog}.${type}.${opt}\" not found"
            fi
         else
            echo "==> file \"test.s\" not found"
         fi
      done
   done
done
```

```
C tests on a collection of compatible loops
C R  Olsen, Fri Jan  3 11:34:40 EST 1992

#define FTIME etime

#ifdef ORIG_TST
      TYPE A(3001), B(3001), C(3001), D(3001),
     +    E(3001), F(3001)
#endif
#ifdef UNROLL1_TST
      TYPE A(3001), B(3001), C(3001), D(3001),
     +    E(3001), F(3001)
#endif
#ifdef UNROLL3_TST
      TYPE A(3001), B(3001), C(3001), D(3001),
     +    E(3001), F(3001)
#endif
#ifdef FUSED_TST
      TYPE A(3001), B(3001), C(3001), D(3001),
     +    E(3001), F(3001)
#endif
#ifdef CONTR_TST
      TYPE A, B, C, D, E, T, F(3001)
#endif
#ifdef PIPE_TST
      TYPE A, B, C, D, E, T, F(3001)
#endif
#ifdef PIPEBUF_TST
      TYPE A, B, BB, C, D, E, T, F(3001)
#endif
      INTEGER I, J, N
C, BASE, NMAX, INCR, ITER
      REAL BEGIN(2), END(2), OVERHEAD, TIME

C     BASE = 60
C     NMAX = 3000
      INCR = 60
C     ITER = 50
C compute timer overhead
      CALL FTIME(BEGIN)
      CALL FTIME(END)
      OVERHEAD = END(1) + END(2) - BEGIN(1) - BEGIN(2)
      OVERHEAD = MCLOCK()
C note that clock resolution on the Sun 4/490 is 1/10 of a second

C print report headings
      WRITE(6,5) BASE, NMAX, INCR
5     FORMAT(' BASE:',I6,' NMAX:',I4,' INCR:',I4)
      WRITE(6,7)
7     FORMAT(' VECTOR LENGTH   TIME (USEC)')

C   run the loop calculation over a range of vector sizes
      DO 80 N=BASE, NMAX, INCR
C  the [illegible] iterations of the loop calculation
      CALL FTIME(BEGIN)
C     BEGIN = MCLOCK()
      DO 70 J=1, ITER

C the loop-cluster calculation begins here
#ifdef ORIG_TST
      DO 10 I=1, N
10       A(I) = I
      DO 20 I=1, N
20       B(I) = N - I
      DO 30 I=1, N
30       C(I) = A(I) + B(I)
      DO 40 I=1, N
40       D(I) = C(I) + 99
      DO 50 I=1, N
50       E(I) = C(N+1-I) + B(N+1-I)
      DO 60 I=1, N
60       F(I+1) = F(I) + D(N+1-I) + E(I)
```

```
#endif
#ifdef UNROLL1_TST
      DO 10 I=1, N, 2
         A(I) =  I
10       A(I+1) =  I+1
      DO 20 I=1, N, 2
         B(I) = N - I
20       B(I+1) = N - I-1
      DO 30 I=1, N, 2
         C(I) = A(I) + B(I)
30       C(I+1) = A(I+1) + B(I+1)
      DO 40 I=1, N, 2
         D(I) = C(I) + 99
40       D(I+1) = C(I+1) + 99
      DO 50 I=1, N, 2
         E(I) = C(N+1-I) + B(N+1-I)
50       E(I+1) = C(N-I) + B(N-I)
      DO 60 I=1, N, 2
         F(I+1) = F(I) + D(N+1-I) + E(I)
60       F(I+2) = F(I+1) + D(N-I) + E(I+1)
#endif
#ifdef UNROLL3_TST
      DO 10 I=1, N, 4
         A(I) =  I
         A(I+1) =  I+1
         A(I+2) =  I+2
10       A(I+3) =  I+3
      DO 20 I=1, N, 4
         B(I) = N - I
         B(I+1) = N - I-1
         B(I+2) = N - I-2
20       B(I+3) = N - I-3
      DO 30 I=1, N, 4
         C(I) = A(I) + B(I)
         C(I+1) = A(I+1) + B(I+1)
         C(I+2) = A(I+2) + B(I+2)
30       C(I+3) = A(I+3) + B(I+3)
      DO 40 I=1, N, 4
         D(I) = C(I) + 99
         D(I+1) = C(I+1) + 99
         D(I+2) = C(I+2) + 99
40       D(I+3) = C(I+3) + 99
      DO 50 I=1, N, 4
         E(I) = C(N+1-I) + B(N+1-I)
         E(I+1) = C(N-I) + B(N-I)
         E(I+2) = C(N-1-I) + B(N-1-I)
50       E(I+3) = C(N-2-I) + B(N-2-I)
      DO 60 I=1, N, 4
         F(I+1) = F(I) + D(N+1-I) + E(I)
         F(I+2) = F(I+1) + D(N-I) + E(I+1)
         F(I+3) = F(I+2) + D(N-1-I) + E(I+2)
60       F(I+4) = F(I+3) + D(N-2-I) + E(I+3)
#endif
#ifdef FUSED_TST
      DO 10 I=1, N
         A(N-I+1) = N-I+1
         B(N-I+1)= I-1
         C(N-I+1) = A(N-I+1) + B(N-I+1)
         D(N-I+1) = C(N-I+1) + 99
         E(I) = C(N+1-I) + B(N+1-I)
10       F(I+1) = F(I) + D(N+1-I) + E(I)
#endif
#ifdef CONTR_TST
      DO 10 I=1, N
         A = N + 1 - I
         B = I - 1
         C = A + B
         D = C + 99
         E = C + B
         T = D + E
10       F(I+1) = F(I) + T
#endif
#ifdef PIPE_TST
```

```
C PROLOG
      A = N+1 - 1
      B = 0
      C = A + B
      D = C + 99
      E = C + B
      T = D + E
      F(2) = F(1) + T

      A = N+1 - 2
      B = 1
      C = A + B
      D = C + 99
      E = C + B
      T = D + E
      F(3) = F(2) + T

      A = N+1 - 3
      B = 2
      C = A + B
      D = C + 99
      E = C + B

      A = N+1 - 4
      B = 3

C BODY
      DO 10 I=5, N
         T = D + E
         F(I-1) = F(I-2) + T
         C = A + B
         D = C + 99
         E = C + B
         A = N+1 - I
10       B = I - 1

C EPILOG
      T = D + E
      F(N+1-1) = F(N+1-2) + T

      C = A + B
      D = C + 99
      E = C + B
      T = D + E
      F(N+1) = F(N+1-1) + T
#endif
#ifdef PIPEBUF_TST
C PROLOG
      A = N+1 - 1
      B = 0
      C = A + B
      BB = B
      D = C + 99
      E = C + BB
      T = D + E

      A = N+1 - 2
      B = 1
      C = A + B
      BB = B
      D = C + 99
      E = C + BB

      A = N+1 - 3
      B = 2
      C = A + B
      BB = B

      A = N+1 - 4
      B = 3

C BODY
      DO 10 I=5, N-1
```

```
            F(I-3) = F(I-4) + T
            T = D + E
            D = C + 99
            E = C + BB
            C = A + B
            BB = B
            A = N+1 - I
10          B = I - 1

C EPILOG
      F(N+1-4) = F(N+1-5) + T

      T = D + E
      F(N+1-3) = F(N+1-4) + T

      D = C + 99
      E = C + BB
      T = D + E
      F(N+1-2) = F(N+1-3) + T

      C = A + B
      BB = B
      D = C + 99
      E = C + BB
      T = D + E
      F(N+1-1) = F(N+1-2) + T

      A = N+1 - N
      B = N - 1
      C = A + B
      BB = B
      D = C + 99
      E = C + BB
      T = D + E
      F(N+1) = F(N+1-1) + T
#endif
70    CONTINUE
      CALL ETIME(END)
      TIME = END(1) + END(2) - BEGIN(1) - BEGIN(2) - OVERHEAD
80    WRITE(6,90) N, TIME * 1000000 / ITER
C     ENDT = MCLOCK()
C     TIME = ENDT - BEGIN - OVERHEAD
C write vector length and time (in microseconds)
C 80    WRITE(6,90) N, TIME * 10000 / ITER
90    FORMAT(I10,F17.1)
      END
```

```
C tests on a collection of non-compatible loops
C R  Olsen, Sun Dec 15 13.43:15 EST 1991

#define ETIME etime

#ifdef ORIG_TST
      TYPE A(3000), B(3000), C(3000), D(3000),
     +   E(3000), F(3000), G(3000), H(3000)
#endif
#ifdef ORIGFIX_TST
      TYPE A(3000), B(3000), C(3000), D(3000),
     +   E(3000), F(3000), G(3000), H(3000), b, e
#endif
#ifdef NAIVE_TST
      TYPE A(3000), B(3000), C(3000), D(3000),
     +   E(3000), F(3000), G(3000), H(3000)
#endif
#ifdef LSNAIVE_TST
      TYPE A(3000), B(3000), C(3000),
     +   E(3000), F(3000), G(3000), H(3000), a, b, d, e
#endif
#ifdef EFF_TST
      TYPE A(3000), B(3000), C(3000), D(3000),
     +   E(3000), F(3000), G(3000), H(3000)
#endif
#ifdef MOREFF_TST
      TYPE A(3000), E(3000), H(3000), b, c, d, e, f, g
#endif
      INTEGER I, J, N
C,    BASE, NMAX, INCR, ITER
      REAL BEGIN(2), END(2), OVERHEAD, TIME

C     BASE = 60
      NMAX = 1000
C     INCR = 60
C     ITER = 50
C compute timer overhead
      CALL ETIME(BEGIN)
      CALL ETIME(END)
      OVERHEAD = END(1) + END(2) - BEGIN(1) - BEGIN(2)
C note that clock resolution on the Sun 4/490 is 1/10 of a second

C print report headings
      WRITE(6,5) BASE, NMAX, INCR
5     FORMAT('  BASE ',I6,' NMAX:',I4,' INCR:',I4)
      WRITE(6,7)
7     FORMAT('  VECTOR LENGTH   TIME (USEC)')

C time the loop calculation over a range of vector sizes
      DO 80 N=BASE, NMAX, INCR
C time 1000 iterations of the loop calculation
      CALL ETIME(BEGIN)
      DO 70 J=1, ITER

C the loop-cluster calculation begins here
#ifdef ORIG_TST
      DO 10 I=1, N
10       A(I) = I
      DO 20 I=1, N
         B(I) = A(I) * 2 + 3
20       C(I) = B(I) + 99
      DO 30 I=1, N
30       D(I) = A(N-I+1) + 6
      DO 40 I=1, N
40       E(I) = B(I) + C(I) * D(I)
      DO 50 I=1, N
         F(I) = E(I) * 4 + 2
50       G(I) = E(I) * 8 - 3
      DO 60 I=1, N
60       H(I) = F(I) + G(I) * E(N-I+1)
#endif
#ifdef ORIGFIX_TST
      DO 10 I=1, N
```

```
10       A(I) =  I
      DO 20 I=1, N
         b = A(I) * 2 + 3
         B(I) = b
20       C(I) = b + 99
      DO 30 I=1, N
30       D(I) = A(N-I+1) + 6
      DO 40 I=1, N
40       E(I) = B(I) + C(I) * D(I)
      DO 50 I=1, N
         e = E(I)
         F(I) = e * 4 + 2
50       G(I) = e * 8 - 3
      DO 60 I=1, N
60       H(I) = F(I) + G(I) * E(N-I+1)
#endif
#ifdef NAIVE_TST
      DO 10 I=1, N
         A(I) =  I
         B(I) = A(I) * 2 + 3
10       C(I) = B(I) + 99
      DO 20 I=1, N
         D(I) = A(N-I+1) + 6
         E(I) = B(I) + C(I) * D(I)
         F(I) = E(I) * 4 + 2
20       G(I) = E(I) * 8 - 3
      DO 30 I=1, N
30       H(I) = F(I) + G(I) * E(N-I+1)
#endif
#ifdef LSNAIVE_TST
      DO 10 I=1, N
         a =  I
         b = a * 2 + 3
         A(I) = a
         B(I) = b
10       C(I) = b + 99
      DO 20 I=1, N
         d = A(N-I+1) + 6
         e = B(I) + C(I) * d
         E(I) = e
         F(I) = e * 4 + 2
20       G(I) = e * 8 - 3
      DO 30 I=1, N
30       H(I) = F(I) + G(I) * E(N-I+1)
#endif
#ifdef EFF_TST
      DO 10 I=1, N
10       A(I) =  I
      DO 20 I=1, N
         B(I) = A(I) * 2 + 3
         C(I) = B(I) + 99
         D(I) = A(N-I+1) + 6
20       E(I) = B(I) + C(I) * D(I)
      DO 30 I=1, N
         F(I) = E(I) * 4 + 2
         G(I) = E(I) * 8 - 3
30       H(I) = F(I) + G(I) * E(N-I+1)
#endif
#ifdef MOREFF_TST
      DO 10 I=1, N
10       A(I) =  I
      DO 20 I=1, N
         b = A(I) * 2 + 3
         c = b + 99
         d = A(N-I+1) + 6
20       E(I) = b + c * d
      DO 30 I=1, N
         e = E(I)
         f = e * 4 + 2
         g = e * 8 - 3
30       H(I) = f + g * E(N-I+1)
#endif
70    CONTINUE
```

```
      CALL ETIME(END)
      TIME = END(1) + END(2) - BEGIN(1) - BEGIN(2) - OVERHEAD
C Write vector length and time (in microseconds)
      WRITE(6,90) N, TIME * 1000000 / ITER
90    FORMAT(I10,F17.1)
      END
```

```
C  MOREFF: PARTITIONING TEST FOR NON-COMPATIBLE LOOPS

      INTEGER N, I
      REAL*8 A(3000), E(3000), H(3000), B1, C, D, E1, F, G
      REAL*8  BEGIN, END, OVERHEAD, TIME, MFLOPS

C  INITIALIZE INPUTS
      BASE = 60
      NMAX = 3000
      INCR = 60

      CALL CPUTIME(BEGIN,RTNCODE)
      CALL CPUTIME(END,RTNCODE)
      OVERHEAD = END - BEGIN

      WRITE(6,5)
5     FORMAT('  VECTOR LENGTH   TIME (USEC)')

C  TIME THE COLLECTION FOR VARIOUS VECTOR SIZES
      DO 70 N = BASE, NMAX, INCR
      CALL CPUTIME(BEGIN,RTNCODE)

      DO 10 I=1, N
10       A(I) =  I
      DO 20 I=1, N
         B1 = A(I) * 2 + 3
         C = B1 + 99
         D = A(N-I+1) + 6
20       E(I) = B1 + C * D
      DO 30 I=1, N
         E1 = E(I)
         F = E1 * 4 + 2
         G = E1 * 8 - 3
30       H(I) = F + G * E(N-I+1)

      CALL CPUTIME(END,RTNCODE)
      TIME = END - BEGIN - OVERHEAD
      WRITE(6,80) N, TIME
70    CONTINUE
80    FORMAT(I9,F17.1)
      END
```

```
/************************** original collection ****************/


main ()
{  int i;
   TYPE A[N+1], B[N+1], C1[N+1], C2[N+1], C3[N+1], C4[N+1];
   float j;

   C4[0] = 0;
   for (i=1; i<=N; i++)
      A[i] =  i;
   for (i=1; i<=N; i++)
      B[i] = N - i;
   for (i=1; i<=N; i++)
      C1[i] = A[i] + B[i];
   for (i=1; i<=N; i++)
      C2[i] = C1[i] + 99;
   for (i=1; i<=N; i++)
      C3[i] = C1[N+1-i] + B[N+1-i];
   for (i=1; i<=N; i++)
      C4[i] = C4[i-1] + C2[N+1-i] + C3[i];

   for (i=1; i<=N; i++)
      printf("%d\n",C4[i]);

   j = 0.34;
 } /* main */

/************************** unrolled once ****************/


main ()
{  int i;
   TYPE A[N+1], B[N+1], C1[N+1], C2[N+1], C3[N+1], C4[N+1];

   C4[0] = 0;
   for (i=1; i<=N; i+=2)
   {  A[i] =  i;
      A[i+1] = i+1;
   }
   for (i=1; i<=N; i+=2)
   {  B[i] = N - i;
      B[i+1] = N - i - 1;
   }
   for (i=1; i<=N; i+=2)
   {  C1[i] = A[i] + B[i];
      C1[i+1] = A[i+1] + B[i+1];
   }
   for (i=1; i<=N; i+=2)
   {  C2[i] = C1[i] + 99;
      C2[i+1] = C1[i+1] + 99;
   }
   for (i=1; i<=N; i+=2)
   {  C3[i] = C1[N+1-i] + B[N+1-i];
      C3[i+1] = C1[N+2-i] + B[N+2-i];
   }
   for (i=1; i<=N; i+=2)
   {  C4[i] = C4[i-1] + C2[N+1-i] + C3[i];
      C4[i+1] = C4[i] + C2[N-i] + C3[i+1];
   }

   for (i=1; i<=N; i++)
      printf("%d\n",C4[i]);

} /* main */

/************************** unrolled three times ****************/

main ()
```

```
{  int i;
   TYPE A[N+1], B[N+1], C1[N+1], C2[N+1], C3[N+1], C4[N+1];

   C4[0] = 0;
   for (i=1; i<=N; i+=4)
   {  A[i] = i;
      A[i+1] = i + 1;
      A[i+2] = i + 2;
      A[i+3] = i + 3;
   }
   for (i=1; i<=N; i+=4)
   {  B[i] = N - i;
      B[i+1] = N - i - 1;
      B[i+2] = N - i - 2;
      B[i+3] = N - i - 3;
   }
   for (i=1; i<=N; i+=4)
   {  C1[i] = A[i] + B[i];
      C1[i+1] = A[i+1] + B[i+1];
      C1[i+2] = A[i+2] + B[i+2];
      C1[i+3] = A[i+3] + B[i+3];
   }
   for (i=1; i<=N; i+=4)
   {  C2[i] = C1[i] + 99;
      C2[i+1] = C1[i+1] + 99;
      C2[i+2] = C1[i+2] + 99;
      C2[i+3] = C1[i+3] + 99;
   }
   for (i=1; i<=N; i+=4)
   {  C3[i] = C1[N+1-i] + B[N+1-i];
      C3[i+1] = C1[N-i] + B[N-i];
      C3[i+2] = C1[N-1-i] + B[N-1-i];
      C3[i+3] = C1[N-2-i] + B[N-2-i];
   }
   for (i=1; i<=N; i+=4)
   {  C4[i] = C4[i-1] + C2[N+1-i] + C3[i];
      C4[i+1] = C4[i] + C2[N-i] + C3[i+1];
      C4[i+2] = C4[i+1] + C2[N-1-i] + C3[i+2];
      C4[i+3] = C4[i+2] + C2[N-2-i] + C3[i+3];
   }

   for (i=1; i<=N; i++)
      printf("%d\n",C4[i]);

} /* main */

/************************** fused loops ****************/


main ()
{  int i;
   TYPE A[N+1], B[N+1], C1[N+1], C2[N+1], C3[N+1], C4[N+1];

   C4[0] = 0;
   for (i=1; i<=N; i++)
   {  A[N-i+1] =  N-i+1;
      B[N-i+1] = i-1;
      C1[N-i+1] = A[N-i+1] + B[N-i+1];
      C2[N-i+1] = C1[N-i+1] + 99;
      C3[i] = C1[N+1-i] + B[N+1-i];
      C4[i] = C4[i-1] + C2[N+1-i] + C3[i];
   }

   for (i=1; i<=N; i++)
      printf("%d\n",C4[i]);

} /* main */

/************************ fused and contracted loops ***************/
```

```
main ()
   int i;
   TYPE a, b, c1, c2, c3, C4[N+1];

   C4[0] = 0;
   for (i=1; i<=N; i++)
   {  a = (N+1) - i;
      b = i - 1;
      c1 = a + b;
      c2 = c1 + 99;
      c3 = c1 + b;
      C4[i] = C4[i-1] + c2 + c3;
   }

   for (i=1; i<=N; i++)
      printf("%d\n",C4[i]);

} /* main */

/************************* software pipelined loops ***************/


main ()
   int i;
   TYPE a, b, c1, c2, c3, d, C4[N+1];

 /* prolog */
   [illegible]

   a = (N+1) - 1;
   [illegible]

   c1 = a + b;
   c2 = c1 + 99;
   c3 = c1 + b;
   d = c2 + c3;
   [illegible]

   a = (N+1) - 2;
   [illegible]
      c1 = a + b;
      c2 = c1 + 99;
      c3 = c1 + b;
   [illegible]
   [illegible]

   a = (N+1) - 3;
   [illegible]
      c1 = a + b;
   [illegible]
   [illegible]

   a = (N+1) - 4;
   [illegible]

 /* body */
   [illegible]
   [illegible]
   [illegible]
   [illegible]
   [illegible]
   [illegible]
   [illegible]

 /* epilog */
   [illegible]
   [illegible]
   [illegible]
```

```
   C4[N] = C4[N-1] + d;

   for (i=1; i<=N; i++)
      printf("%d\n",C4[i]);

} /* main */

/****************** software pipelined loop with buffer ***************/


main ()
{  int i;
   TYPE a, b, bb, c1, c2, c3, d, C4[N+1];

/* prolog */
   C4[0] = 0;

   a = (N+1) - 1;
   b = 0;
   c1 = a + b;
   bb = b;
   c2 = c1 + 99;
   c3 = c1 + bb;
   d = c2 + c3;

   a = (N+1) - 2;
   b = 1;
   c1 = a + b;
   bb = b;
   c2 = c1 + 99;
   c3 = c1 + bb;

   a = (N+1) - 3;
   b = 2;
   c1 = a + b;
   bb = b;

   a = (N+1) - 4;
   b = 3;

/* body */
   for (i=5; i<N; i++)
      C4[i-4] = C4[i-5] + d;
      d = c2 + c3;
      c2 = c1 + 99;
      c3 = c1 + bb;
      c1 = a + b;
      bb = b;
      a = (N+1) - i;
      b = i - 1;
      /* for i */

 /* epilog */
   C4[N-4] = C4[N-5] + d

   d = c2 + c3
   C4[N-3] = C4[N-4] + d

   c2 = c1 + 99;
   c3 = c1 + bb
   d = c2 + c3
   C4[N-2] = C4[N-3] + d

   c1 = a + b
   bb = b
   c2 = c1 + 99;
   c3 = c1 + bb;
   d = c2 + c3;
   C4[N-1] = C4[N-2] + d

   a = (N+1) - [illegible]
   b = [illegible] - 1
```

Appendix B

SPARC Architecture

In this appendix we review the basic features of Sun's SPARC architecture and f77 FORTRAN compiler as they pertain to the SPARC Server 4/490, the scalar RISC workstation used during the testing of collective loop transformations described in Chapter 6. In subsequent appendices we provide similar information regarding the remaining architectures and compilers used during our experiments. In this regard, neither this appendix nor the ones that follow are intended to be comprehensive. Rather, the purpose of these appendices is merely to introduce the salient features of each architecture and compiler in sufficient detail to permit understanding of the results presented in Chapter 6. With this objective in mind, we begin with a brief description of the SPARC architecture.

B.1 Base Architecture

SPARC, an acronym for Scalable Processor Architecture, is an open architecture specification which has now been implemented by several manufacturers. The features of this architecture are similar in many ways to the DLX architecture described in Appendix E [HP90]. The architecture itself derives many of its design features from the earlier Berkeley RISC II and SOAR architectures: register windows, delayed branches, delayed loads with hardware interlocks, a floating point coprocessor, and a few special instructions to support tagged data [Muc88].

The architecture has three sets of registers: global integer registers, global floating point registers, and windowed integer registers. The windowed registers are further

Table B.1: SPARC Register Names

primary identifier	alternate identifier	purpose
r24–r31	i0–i7	window ins
r16–r23	l0–l7	window locals
r8–r15	o0–o7	window outs
r0–r7	g0–g7	globals
sp	o6	stack pointer
fp	i6	frame pointer
	07	return address
g0		zero value

divided into three subsets: ins, outs, and locals. The windowed registers are used in conjunction with procedures (see Table B.1). A `save` instruction, executed as part of a procedure prologue, changes the machine's interpretation of register numbers so that the calling procedure's outs become the called procedure's ins, and a new set of locals and outs become available. In addition, the save allocates a new stack frame by setting a new stack pointer from the old one. A corresponding `restore` instruction in the called procedure's epilog restores the caller's register number interpretation and reduces the stack. When all of the register sets are filled and another one is required, the system traps, and the oldest window is spilled. Conversely the system traps when an attempt is made to restore a previously stored window.[1] By convention global integer and floating point values are referenced using a register offset, register `g0`, which is wired to zero, is used to achieve the effect of absolute addressing.

Computational instructions obtain all of their data from registers or from 13-bit sign extended immediate fields in instructions and put their results in registers, with one exception, `sethi`. The sethi instruction constructs 32-bit constants and addresses for access to global data. It loads an immediate 22-bit constant into the high end of a register and puts zeros in the other ten bits. For example to load the word at address `loc` into `i2` the following code sequence is used:[2]

[1] Much information exists regarding the use and benefit of register windows, but this material is largely irrelevant to our particular research [DS90, KW88, HP90, UBF+81].

[2] In SPARC assembly language source operands precede the result operand, and the first operand of a three operand instruction is always a register name. Each register reference is preceded by a percent sign (`%`), the `%hi()` operator extracts the high-order 22 bits of its operand, and the `%lo()`

```
sethi %hi(loc),%r1
ld    [%r1+%lo(loc)],%r2
```

Only load and store instructions access memory. The load and store instructions have two addressing modes: one that adds the contents of two integer registers and one that adds the contents of an integer register and a 13 bit signed immediate.

SPARC provides no integer multiply, divide, or remainder instructions, so these operations are constructed from more elementary instructions. It does have a multiply step instruction **mulscc**, however, it is not used to multiply variables by constants. Instead, it does multiplication by constants known at compile time, using sequences of shifts and adds. For example, a source-level multiplication by thirty is translated to

```
sll %o2,1,%o2         ! 2 * x -> x
sll %o2,4,%o3         ! 16 * x -> y
sub %o3,%o2,%o2       ! y - x -> x
```

SPARC requires that data be aligned according to its size, and referencing data that is not properly aligned will cause a trap.

B.2 The SPARC f77 FORTRAN Compiler

Sun uses the same code generation and optimization technology for all of its production compilers for SPARC, and in this section we describe the particular features which affect the quality of code generated by these compilers. Of particular importance in this section are the native optimizations applied by the f77 FORTRAN compiler which affect the test results reported in Chapter 6.

The compiler has four levels of optimization, besides the level of no optimization. These levels of optimization are:

O1: peephole optimization,

O2: peephole and global optimization, excluding expressions involving global variables and pointers,

O3: lower-levels of optimization along with optimization of expressions involving global variables but with worst-case assumptions being made regarding pointers and

operator extracts the low-order 13 bits. Memory references are bracketed.

O4: all optimizations with pointers being traced during pointer analysis.

Levels O3 and O4 almost always produce the same code, except for situations such as in which C programs take the addresses of local variables and then dereference them

Global optimization refers to optimizations which operate on whole procedures as opposed to basic blocks, the compiler does not do interprocedural analysis (circa 1988) The optimizations the compiler performs are loop-invariant code motion, induction variable strength reduction, common subexpression elimination (local and global), copy propagation (local and global), register allocation using modified graph coloring, dead code elimination, loop unrolling, and tail-recursion elimination A separate pass handles code in-lining The peephole optimizer eliminates unnecessary jumps, eliminates redundant loads and stores, deletes unreachable code, does loop inversion, utilizes machine idioms, performs register coalescing, performs instruction scheduling, does leaf-routine optimization, performs cross jumping, and handles constant propagation [ASU86]

In addition to the above optimizations, the compiler implements tail-recursion elimination, in line expansion, and loop unrolling. Tail-recursion elimination is important because it helps to prevent costly register window overflows Likewise, in-line expansion also eliminates procedure overhead, but in addition, it creates an opportunity for further optimization by such means as peephole optimization Loop unrolling is applied to loops which satisfy the following four conditions *1)* they contain only a single basic block, that is, they contain only straight-line code, *2)* they generate at most forty triples of Sun intermediate-representation code, *3)* they contain floating-point operations, and *4)* they have simple loop control. Loops which satisfy these conditions are unrolled once, a compiler option allows higher levels of unrolling

Branch execution takes effect after a one instruction delay, with the option to not execute the delay slot instruction Load instructions can overlap other instructions provided the subsequent instruction does not use the value being loaded Likewise, SPARC specification allows concurrent execution of integer and floating-point instructions, the exact amount of concurrency being implementation dependent The instruction scheduler for a Sun-4 implementation allows one additive and one multiplicative floating-point operation to occur concurrently Therefore, the instruction scheduler is designed to handle as many as four types of operations at once *1)* a branch or load, *2)* an integer instruction other than a branch or load, *3)* an additive floating point operation, and *4)* a multiplicative floating-point operation The actual instruction scheduler used by the SPARC compiler is the one developed by Gibbons and Muchnick (see Chapter 5) [GM86, Muc88].

Multiplication of variables by variables and all divisions and remainders other than by powers of two are computed by calling special leaf routines. As an example, the SPARC performance for integer multiplication is shown below.

length (in bits)	cycles
1–4	18
5–8	25
9–12	33
13–16	41
17–32	60

The preceding cycle counts are for nonnegative multipliers; negative multipliers require one more cycle for operands up to sixteen bits and up to four more cycles for longer operands.

Appendix C

IBM RISC System/6000

This appendix summarizes several key features of the IBM RISC System/6000 and its native xlf FORTRAN compiler, used to evaluate the effectiveness of collective loop transformations described in Chapter 6. The RS/6000, as it is known, comes in a wide range of models, from a low end Model 320, to a high end, Model 930. The particular model used in our experiments was a Model 550. Although minor architectural differences exist between these models, the material presented in this appendix generally applies to all models, unless otherwise noted. The information which follows was extracted primarily from IBM manuals on RISC System/6000 technology and performance tuning [Mis90, Bel90]. As in the previous appendix, we begin with an architecture description and follow it with a brief compiler description.

C.1 Base Architecture

The IBM RISC System/6000 employs a superscalar processor which is distributed across several semi custom chips [BW90, GO90]: an Instruction cache Unit (ICU), a Fixed-Point Unit (FXU), a Floating-Point Unit (FPU), four Data Cache Units (DCUs), a Storage Control Unit (SCU), an Input/Output Interface unit, and a clock chip. The basic organization of the RS/6000 is shown in Figure C.1.

The ICU contains a branch control unit, condition code registers, instruction cache, and instruction translation lookaside buffer (TLB), along with the central control hardware for interrupts. Fixed- and floating-point instructions go to instruction buffers in the FXU and FPU where the instructions can be concurrently executed.

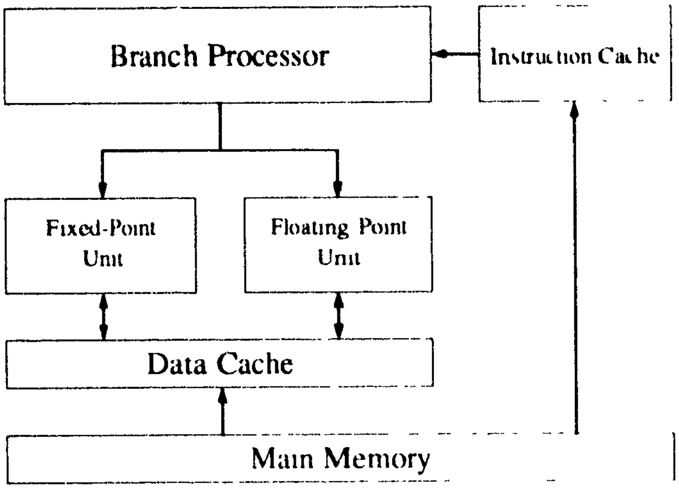


Figure C.1: The IBM RISC System/6000 Architecture

The instruction buffers themselves can each contain up to twelve entries. Besides the fixed-point arithmetic unit, the FXU contains the fixed point registers, segment registers, and data TLB. The FXU decodes and executes all fixed point instructions and floating-point load and store instructions. In addition the FXU performs page table updates for the two TLBs and page-table updates. The FPU also contains, of course, the machine's floating-point registers, as well as six rename registers and two divide registers. In addition, a five entry pending store queue and a four entry store data queue in the FPU enable the FXU to execute floating point store operations before the FPU produces the data. The cycle time of the processor varies depending upon the model, but for the Model 550, which was used for the tests the clock rate 41M Hertz. At this rate the processor is able to execute floating point operations at a peak rate anywhere from 10 60 MFLOPS.

Among the unique features of the RS/6000 architecture are *1)* multiple sets of condition codes that are managed by the compiler much like general purpose registers in a load/store architecture, *2)* update forms of load and store instructions update the contents of an address register with the results of a displacement computation for the address of the next operand, and *3)* compound operations which combine two arithmetic operations and execute them as a single operation. Theoretically the processor can execute four instructions per cycle a branch, a logical compare, a fixed-point instruction, and a floating-point instruction. Moreover, the floating-point instruction can be a compound multiply/add instruction (FMA), for a a total of five instructions per cycle.[1] The condition code sets allow several condition register altering operations to be underway concurrently and increasing parallelism between fixed-point and floating-point units. This allows branches to often be fully overlapped with computation, giving the effect of a zero-cycle branch delay.

A multiply instruction takes three to five cycles, floating-point divides can take nineteen to twenty cycles, and a terminating branch which is not a BCT/F can take about six cycles. Compound instructions use the same functional units as single operation instructions, but the target of a compound instruction is not available as the input to another multiply/add instruction for two cycles, therefore, a series of compound instructions would issue every other cycle, rather than every cycle as normally would be the case.

The processor has 32 fixed point registers and 32 floating-point registers, and all arithmetic is done to and from these registers. A floating-point load instruction is done by the fixed-point unit and therefore can be done concurrently with a floating-point instruction, provided the floating-point instruction does not depend on the value being loaded. Stores are also done by the fixed-point unit, but since it does interact with the floating-point unit it cannot be executed concurrently, delaying the floating-point unit by one cycle. All floating-point arithmetic in an RS/6000 is done in double precision, and therefore, single-precision arithmetic is actually slower than double precision because of the extra instructions needed to convert the double-precision results back to single precision. The advantage of using single-precision numbers, on the other hand, is that cache and TLB misses should occur on average half as often and twice as many numbers can be held in each cache line and memory page.

The system has a four word memory bus, a four-word instruction-fetch bus from the instruction cache, a one word data bus between the fixed-point unit and data cache and a two word data bus between the floating-point unit and data cache. The instruction cache unit, located on the ICU, is a two way set-associative 8K-byte cache with a line size of 64 bytes, and the instruction TLB is a 32-entry, two-way set-associative cache.

The data cache employs a four-way associative, 64K-byte data cache having 128-byte lines. Each of the four 128-line sets of 16K bytes corresponds to a 16K-byte

[1] There are actually four compound instructions: multiply/add, multiply/subtract, negative multiply/add, and negative multiply/subtract.

memory page. Therefore, any particular line from memory can go into any one of only four lines in the cache. Cache lines are retrieved from memory line at a time, but the particular word referenced within the requested line is transferred to register first. Both the loading of cache lines and the copy back of lines to memory are buffered to minimize cache-line replacement time. The cache uses a Least Recently Used (LRU) line-replacement policy and a copyback principle such that only modified lines are written back to memory at the time the respective lines are expelled from the cache. Loads from cache to register take one cycle, but if the requested word is not present in the cache there is a delay of eight cycles, from then on, subsequent requests for words within the requested line are satisfied in one cycle, as in the previous case. To support this high transfer rate, the system uses either a 64 bit or 128 bit interface to the machine's four-way interleaved memory.

The Translation Lookaside Buffer (TLB) is a 128 entry, 512K byte, two way set associative cache that retains mapping information between virtual and real addresses for the most recently requested pages. If the virtual address of a 1K page is held in the TLB, there is no delay in accessing the requested page, if it is not there, there is a delay of 32 cycles while the mapping information is fetched from the respective page map table. For sequentially accessed data, the cache miss overhead is much higher than the TLB overhead. For example, for double precision data (eight bytes per number) a cache miss causing an eight cycle delay would occur every sixteen (128/8) data items, whereas, a TLB miss causing a thirty two cycle delay would occur only every 512 items (4K/8). However, were the stride 4K bytes, a cache miss and a TLB miss would occur for each item requested, and therefore, TLB overhead in this case would dominate.

C.2 The xlf FORTRAN Compiler

Although the XL Compilers are a new line of compilers, their origins are with the PL.8 compiler developed for IBM's 801 minicomputer [AH82, OHM+90, Rad82]. The particular FORTRAN version we describe here is an early version, ver. 1.01, which has since been upgraded [IBM90].

The xlf compiler has two levels of optimization: `NOOPT`, or no optimization, and `OPT`. The optimizations performed by xlf are, using IBM terminology: value numbering, straightening, common subexpression elimination, code motion, reassociation and strength reduction, constant propagation, store motion, dead store elimination dead code elimination, in lining, global register allocation and instruction scheduling. Value number is IBM's term for local constant propagation, local subexpression elimination, and instruction folding. IBM uses the term "straightening" to refer to the rearrangement of program code to minimize branching logic and to combine physically separate blocks of code. The term "code motion" is used to refer to loop-invariant code removal from within a loop. Reassociation is the rearranging of a sequence of calculations in a subscript expression so as to produce additional candidates for common subexpression elimination. Store motion refers to the movement of store instructions outside loops.

There are of course several ways the programmer can improve the effectiveness of compiler optimization; a few of the less common ways which apply to xlf are, for example, *1)* using implied DO in input/output statements, such as WRITE(3) (A(I), I=1,100), rather than an explicit DO loop, *2)* using REAL*8 rather than REAL*4, *3)* defining constant operands as local variables rather than global variables (the compiler recognizes only local variables as having a constant value since operands in COMMON or in an argument list can change), *4)* identifying common subexpressions by either putting them at the left of an expression or within parenthesis, e.g.,

A=B*(X*Y*Z)
C=X*Y*Z*D

5) unrolling small loops (unrolling is an option in later versions of the compiler), *6)* avoiding integer to floating-point conversions, and *7)* using temporaries to add negative constants rather than subtracting positive constants within an expression.

Appendix D

IBM 3090/VF

The purpose of this appendix is to describe the characteristics of a vector processor which differentiates it from the architectures described in the previous appendices. In particular, we describe the special features of the IBM 3090/VF and its VS FORTRAN compiler.

D.1 Base Architecture

The 3090 is a high-performance mainframe computer with vector processing capability. Although the machine is targeted towards scientists and engineers, its performance is less than that of most supercomputers [Don88]. The computer itself comes in several models spanning a range of uniprocessors to multiprocessors; the model 180, to which we had access, is a uniprocessor version, but there are several multiprocessor models as well: model 200, 400, and 600, having two, four, and six processors, respectively. From a programmer's point of view, the 3090 is like any other System/370, except that it has additional registers and instructions to support the Vector Facility.

The standard register configuration consists of sixteen 32-bit general purpose registers which can be paired to form eight 64-bit registers and four 64-bit floating point registers. The standard configuration also consists of sixteen 32-bit control registers, registers for a CPU timer and time-of-day clock, and a Program Status Word (PSW) containing the instruction counter and status flags.[1] The 3090, Model 180J, operates

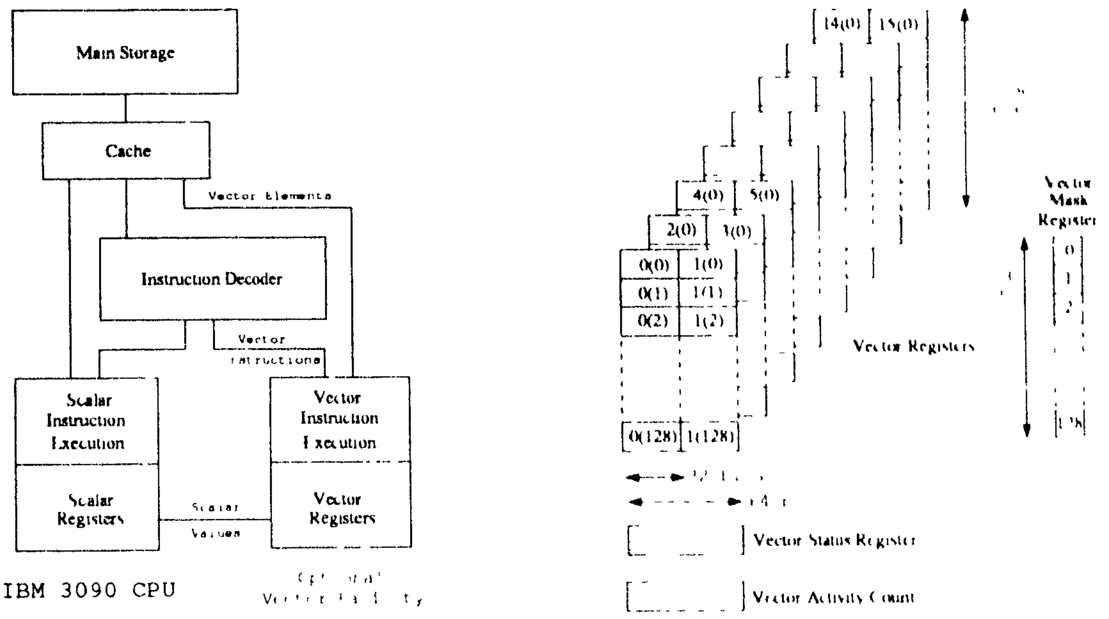
[1] The 3090 implements IBM's System/370-XA architecture, or *extended architecture*, an upgrade

with a cycle time of 14.5 nanoseconds, permitting a scalar peak performance rate of 69 MFLOPS.

The sixteen vector registers, each comprised of 256 four-byte elements, provides 16K bytes of fast memory to the machine. Each of these registers is capable of sustaining two read accesses and one write access per cycle [PMSB88, Tuc86]. At the next level of the memory hierarchy is a 256K-byte cache. On the 3090 Model E, data transfers between cache and memory using 128-byte lines beginning at memory addresses divisible by 128. The cache is 4 way associative and is divided into 128 sets, data from addresses which are separated by multiples of 16 kilobytes compete for space in the same set of four cache lines [LS87]. The cache uses a Least-Recently Used (LRU) line replacement policy and copy-back memory update policy. It supplies data to the vector facility at a rate of eight bytes per cycle, with negligible overhead. The maximum pipelined transfer rate between memory and cache is 16 bytes every two cycles, with a 14 cycle overhead [LS87]. Whereas vector register reuse reduces vector load and store instructions, which take approximately as long to execute as vector arithmetic instructions, cache reuse reduces the time lost in cache-memory data transfer, which is 16 or more cycles per access. Main memory is a virtual memory system which is accessed through two-level page map tables. One main-memory access can be sustained per cycle (either reading or writing). Like virtual memory, expanded memory (or expanded storage, in IBM terminology) is a feature of System/370 XA architecture that was not a part of the original System/370 architecture. With expanded memory, block transfers occur synchronously in 4K-byte pages while the processor waits. These transfers occur directly between expanded memory and main memory at a rate of one quad word every four cycles [Tuc86].

The Vector Facility can be viewed as an additional instruction execution component of the base machine (see Figure D.1). For example, the vector instructions are decoded along with the scalar instructions, but once they are decoded, they are passed to the Vector Facility for execution. As indicated in the figure, all instructions are fetched from cache. In the case of vector instructions, operands come either from one of the Vector Facility's registers, from one of the 3090's scalar registers, or from cache. The Vector Facility has 16 vector registers, 4 floating-point registers, and 16 scalar registers. Each element of a vector is 32-bits wide. A single vector register can hold either a 32 bit integer or short real, and a pair of even-odd registers can hold either a 64-bit vector product of two binary integer vectors or a long real (see Figure D.1b). The *vector-mask register* contains mask bits that can be used to select

which increases the machine's virtual address space to two gigabytes, among other improvements.



a. CPU

b. Vector Facility[2]

Figure D.1: The IBM 3090 CPU and Optional Vector Facility

which elements of the vector register are to be processed and for checking for elements that are zero before a divide operation. The *vector-status register* holds control fields such as a mask-mode bit, a count to control the number of elements within each vector register that are to be processed, an interruption index, and other operating system and application program status. The *vector-activity count* keeps track of the time spent executing vector instructions and is incremented every microsecond while a vector instruction is being executed.

In general, vector instructions are available in one of four formats: *1)* VST: one operand in storage, the other, if any, in a vector register, *2)* VV: all operands in vector registers, *3)* QST: one operand in storage, the other in a scalar register, and *4)* QV: one operand in scalar register; the other, if any, in a vector register. Addition, subtraction, multiplication, and comparison operations are available in twelve versions: one for

[2] Vector register capacity and vector mask register size are model dependent. for the 3090 Model 180E, the capacity and register size are 128 elements and 128 bits, as shown.

each data format binary integer and short and long floating-point formats, division is available in eight formats Altogether, the vector instruction set consists of 171 instructions, as a first order approximation, the time required to execute a vector instruction is 28 cycles (overhead) plus 1 cycle per element thereafter [GRW86]

Vector instructions operate on operands either in registers or in memory, the resultant vector elements always go to a vector register, except when the result of a vector comparison is sent to the vector mask register Vector instructions use the IBM 370's extended (32 bit) virtual addressing [IBM87] Memory is accessed in three ways *1)* sequential addressing, when vector data elements are located in contiguous locations of virtual storage or in locations with a fixed or constant stride *2)* under control of a bit mask for accessing elements in either a vector register and main memory and *3)* indirect addressing, using a set of addresses from a vector register to access data elements that are located randomly in memory

The address of a vector in storage is the address of the first element to be processed: this address is stored in one of the 16 general-purpose registers and as a vector is being processed the address in the general-purpose register is incremented. The number of element positions in storage needed to advance from one vector element to the next is called the *stride* For an $m \times n$ matrix stored in column-major order (the FORTRAN convention) columns are stored contiguously with a length of m and a stride of one, and rows are noncontiguous with a length of n and a stride of m For a row processed in reverse order the stride is minus

Converting a register address to a storage address involves multiplying the vector stride by the register size Since register sizes are a power of two (2^2 for 32-bit registers and 2^3 for 64 bit registers) the increment between successive vector element addresses is automatically computed by the machine by shifting the respective number of bits. For example, if the stride for a floating point vector in the long format is 10 (or 1010 in binary), the corresponding address increment is 80 bytes (or 1010 000)

When a vector is loaded into a register, its elements occupy consecutive register positions, and the number of elements loaded is placed in the *vector count* The count can be any integer from zero up to section size The *vector interruption index* indicates the element in the vector register or registers currently being processed During instruction execution the vector interruption index normally starts at zero advances until it reaches the vector count, and then resets to zero If a vector instruction is interrupted for any reason, the vector interruption index marks the point that was reached, and execution resumes from that point when the instruction is reissued

Vectors which are longer than the number of elements a register can hold are processed in sections. section length varies from model to model, but 256 elements is a common size. Sectioning refers to a technique for processing vectors of any length in sections, where a section is either the number of elements that can be held in a vector register or, for the last section, some number of elements greater than zero but less than the section size of the machine. Sectioning is sometimes called strip mining, referring to the way in which vector sections are stripped away from the original vector, for processing in section size increments. An example of loop sectioning is shown below. To the left is a loop which performs a simple vector addition, and to the right is an equivalent loop, written in FORTRAN, as it might logically be performed.

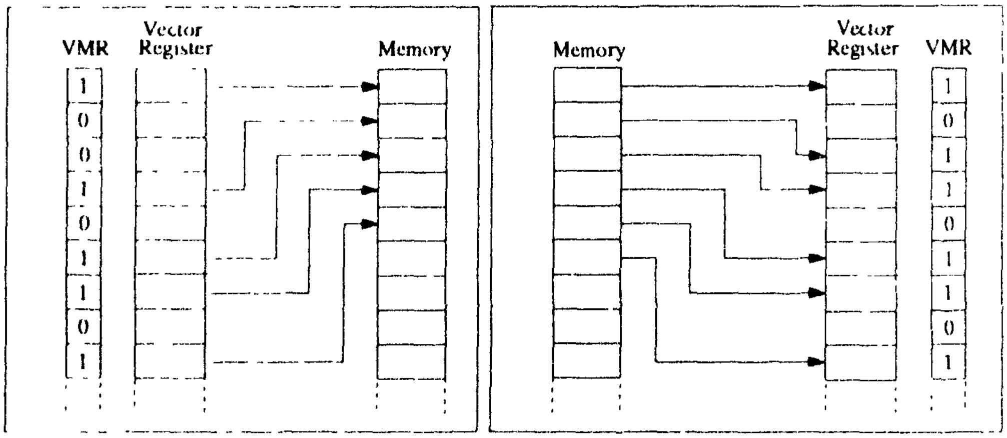
```
      DO 10 I = 1, N
10       C(K) = A(K) + B(K)
```

```
      DO 10 I = 1, N, 128
         DO 10 K = I, I + MIN(N-I,127)
10       C(K) = A(K) + B(K)
```

The way a loop is actually segmented is indicated by the assembly language statements next.

```
      L      G0,N
      LA     G1,A
      LA     G2,B
      LA     G3,C
LOOP  VLVCU  G0
      VLD    V0,G1
      VAD    V0,V0,G2
      VSTD   V0,G3
      BC     2,LOOP
```

First, the general register G0 is given the vector length, and G1–G3 are given the base addresses of the three relevant vectors. The instruction VLVCU (Load Vector Count and Update) loads the vector count with the lesser of the vector length (specified in general register zero) and the section size (again, for the Model J, section size is 256); it reduces the contents of the general register by the number just loaded into the vector count, and finally it sets the condition code of the machine to indicate whether the new general-register contents are zero. The next three instructions are vector floating-point instructions. VLD loads a section of vector A from storage. VAD adds to that section a section of vector B from storage and returns the result to vector register zero. VSTD stores the result into a section of C. And lastly, BC (Branch on Condition) tests the condition code set by VLVCU; if it is 2, general register



a. Compression

b. Expansion

Figure D 2 Vector Compression and Expansion

zero is greater than zero still, and the instruction branches back Each of the vector instructions advances the vector address in the general register to the first element of the next section so that processing can continue directly from one section to the next

The 3090 has several built in accumulation operations The MAXIMUM and MINIMUM operations produce a result which is the maximum or minimum of an entire vector, regardless of section size ACCUMULATE produces the sum of all elements in a vector, whereas, MULTIPLY AND ACCUMULATE produces the sum of the product elements obtained by multiplying a pair of vectors (the inner or dot product) The order of accumulation can affect the rounding error when and where underflow or overflow occur The result of the first phase of an accumulation is a number of partial sums of p elements, where p is the length of the pipeline (e.g., $p = 4$ for the 3090), and partial sums are accumulated using the instruction SUM PARTIAL SUMS Although the resultant accumulation might be different than that of a strictly sequential accumulation, the vector accumulation is duplicatable and can be replicated by means of a corresponding scalar loop

Although many vector operations perform the same arithmetic operation on all elements of a vector, some applications require that an operation be performed on selected elements. Two ways of handling these situations are using conditional arithmetic and performing element extraction. For conditional arithmetic, the operation is performed on all elements of a vector, and results are only returned for the items selected. For element extraction, the elements required for an operation are selected and compressed prior to the operation; following the operation, the results are expanded and returned to their respective initial locations (see Figure D.2). The 3090 performs conditional operations both ways using the vector mask register.

D.2 VS FORTRAN Compiler

In this section we describe the major program analysis capabilities of VS FORTRAN. VS FORTRAN is the optimizing and vectorizing compiler for IBM's System/370 architecture and, in particular, the company's 3090/VF series of computers with their attached Vector Facilities. An important capability of this compiler is its capability to vectorize general purpose engineering and scientific applications written in FORTRAN 77. The compiler performs this function by identifying DO loops within these applications that can and should be vectorized from a performance standpoint, and generating suitable vector machine code for execution by the Vector Facility hardware. Vectorization analysis and code generation are performed by the compiler only when the VECTOR parameter is passed as an optional argument to a so-called "cataloged" compilation procedure, such as VSF2CLG [McG88, IBM83, McG87]. Besides the VECTOR parameter, these cataloged procedures take other parameters to control functions such as the level of optimization, whether compiler directives are applied, the types of output reports prepared consequent to compilation, and alike.

During the compilation process the VS FORTRAN compiler proceeds through four stages of source code analysis: *1)* eligibility analysis, *2)* recurrence detection, *3)* supportability analysis, and *4)* vector execution selection [IBM83].

During the first stage of analysis the compiler looks for situations which might make loops ineligible for vectorization. Several types of situations make a loop ineligible, for example: branches out of a loop, around an inner loop, or backwards within a loop, loops other than DO loops, loops with a loop index or iteration control expression other than INTEGER*4, loops with induction variables mentioned in EQUIVALENCE statements, I/O statements, computed or assigned GO TO statements, subroutine calls, external, non-intrinsic function references, and reference to CHARACTER data. If any of these situations exist, the loop is not vectorized and the remark "UNAN", or unanalyzable, is annotated aside the corresponding lines in the compiler's vector analysis report.[4]

The second phase of vectorization analysis is recurrence analysis. A recurrence is a group of one or more statements forming a dependence cycle. A dependency is said to be *carried* by a loop if the dependencies involved in the recurrence are caused either by the loop or by some loop at a deeper level of nesting. During recurrence analysis the compiler rejects for vectorization any DO loop for which at least one of the following conditions exists: *1)* The loop contains an unbreakable recurrence. Many types of recurrences are breakable, however, as will be described in the next section. *2)* The loop uses an induction variable which modifies inner DO-loop parameters. An induction variable is an INTEGER*4 variable which is incremented (or decremented) by a fixed amount each time a loop iterates. *3)* There are dependences in outer loops which prevent an inner loop with dependencies from being interchanged. Loop interchange is also described in the next section. When a loop cannot be vectorized because of a recurrence, the corresponding lines on the vector analysis report are marked "RECR".

During the third phase of analysis the compiler checks for operations which are supportable by either the compiler or the vector hardware. The types of operations or constructs which are rejected for vector processing are: certain types of LOGICAL fetches or stores, REAL*16 or COMPLEX*32 operations, certain types of noninductive subscripts, certain intrinsic in-line functions; relational expressions that need to be stored (for example, `L=A GE B`), and misaligned data. Lines on the vector analysis report which correspond to unsupported operations or constructs are marked "UNSUP".

The final phase of vectorization analysis is called the vectorization selection stage. During this phase of compilation the compiler selects loops for vectorization based upon cost considerations (in terms of CPU cycles). If the scalar version of a loop takes less time than the vectorized version, a loop will be executed in scalar mode, i.e., without using the Vector Facility hardware, and the corresponding lines on the vector analysis report will be marked "SCAL". If a loop is eligible and cost effective for vectorization but some other eligible loop in the loop nest will yield yet higher performance, the loop will still be executed in scalar mode, but the corresponding lines on the vector analysis report will be marked "ELIG". Otherwise, if the loop is selected for vectorization, the corresponding lines on the vector analysis report will

[4] The vector analysis report is generated by the compiler when the following parameter is passed to the cataloged compilation procedure: `VEC(REP(XLIST)`

be marked "VECT".

D.2.1 Automatic Vectorization

Whenever VS FORTRAN finds a loop which cannot be vectorized because of a recurrence relationship, it automatically tries to break the recurrence using one of several common vectorizing transformations. In addition, the compiler performs other related optimizations to achieve best performance from the Vector Facility hardware. The types of transformations and optimizations performed by the compiler are: loop selection, statement reordering, loop distribution, scalar expansion, IF conversion, use of compound instructions, use of reduction operations, and intrinsic function recognition.

Loop Selection. If the inner loop is not vectorizable, the compiler will look for an outer loop that is. The following loop is an example of such a situation.

```
          DO 10 I = 1, 100
             DO 10 J = 1, 99
10              Z(I,J+1) = Z(I,J) * X(I,J)
```

Statement Reordering. To break backward dependencies, the compiler will often reorder instructions. For example, the compiler will switch the two statements within the loop shown below in order to redirect the flow dependency which exists between elements of array B.

```
          DO 10 I = 2, 100
             A(I) = B(I-1) * 3.0
10           B(I) = C(I) * 3.0
```

Loop Distribution. Statements within loops which are not bound by a dependency relationship are placed in a separate loop so that these statements can be vectorized. The two statements within the loop shown next, for example, will be placed in separate loops by the compiler so that the first statement can be vectorized.

```
          DO 10 I = 1, 99
             A(I) = A(I) + 2.0
10           B(I+1) = B(I) + 2.0
```

Scalar Expansion. The next example shows a code segment which is made vectorizable by using scalar expansion. As a result of this transformation, the scalar variable T is expanded into a temporary array so that the loop can then be vectorized.

```
      DO 10 I = 1, 100
         T = B(I)
         B(I) = A(I)
10       A(I) = T
```

IF Conversion. If there exists a control dependence in the code segment as shown to the left, below, the compiler will conceptually convert it to a data dependence, as shown to the right, a form which can be vectorized. In this instance the compiler takes advantage of the Vector Facility's vector mask register setting it, with the predicate statement, and then using it to select the affected elements, with the subsequent statement (see Section D.1)

Before Conversion

```
   DO 10 K = 1, N
     IF (A(K) .GT. 0.0) GO TO 10
10   A(K) = B(K) + 1.0
```

After Conversion

```
   DO 10 K = 1, N
     LOGIC(K) = .NOT. (A(K) .GT. 0.0)
10   IF (LOGIC(K)) A(K) = B(K) + 1.0
```

Use of Compound Instructions. Some instructions, such as MULTIPLY AND ADD, are performed concurrently in hardware, and loops such as the matrix multiplication, shown below, can take advantage of these instructions to execute at an operation rate approaching the machine's peak performance rate. In this particular instance, the McGill 3090 executed the matrix multiply loop at a rate of 50.8 MFLOPS, compared to the machine's theoretical peak scalar performance rate of 69 MFLOPS.

```
      DO 10 I = 1, 100
         DO 10 J = 1, 100
            C(I,J) = 0.0
            DO 10 K = 1, 100
10             C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

Use of Reduction Operations. The compiler recognizes when certain types of reduction operations are performed, such as *1)* sum of vector elements, *2)* sum of squares of vector elements, and *3)* scalar (inner) product of two vectors. In these situations, machine instructions are generated to take advantage of the special vector hardware designed to perform these operations. The types of statements for which reduction operations are performed are

```
SUM = SUM + A(I)
SUMSQ = SUMSQ + A(I) * A(I)
SUMPRD = SUMPRD + A(I) * B(I)
```

Since the order of the computation in a vectorized reduction operation is different than it is for a scalar reduction, the results of a vector reduction might also be different. Vector summation, for example, is done by adding every n^{th} element and then adding partial sums; whereas, scalar summation is done sequentially.[5]

Intrinsic Function Recognition. The VS FORTRAN compiler also recognizes standard intrinsic elementary transcendental functions and compiles them to calls to the compiler's standard library. Since the compiler knows these routines, it can safely vectorize the loop in which these statements appear, that is, if there are no other loop dependencies to otherwise prevent vectorization.

D.2.2 Additional Factors Affecting Vector Performance

Much of the work required to vectorize an application program is performed automatically by the VS FORTRAN compiler, however, there are programming practices which help the compiler to do a better job of vectorization. *1)* In many cases, vector directives can be used to provide the compiler with information that is essential for efficient vectorization. Three such directives are `ASSUME COUNT`, `IGNORE` and `PREFER`. `ASSUME COUNT` specifies a value that is to be used for vector cost analysis when a DO loop iteration count cannot be determined at compile time. `IGNORE` instructs the compiler to ignore specified dependencies in a DO loop, and `PREFER` specifies that a DO loop be processed in vector mode, if eligible, or scalar mode, regardless of decisions made as a result of vector cost analysis. *2)* Direct subscript expressions should be used rather than indirect expressions accessed through temporaries. For example,

```
X(I+1) = A     is preferable to     J = I + 1
                                    X(J) = A
```

Whereas loop unrolling is an effective optimization to reduce loop overhead in scalar code, it has an adverse impact upon vectorized code (see Chapter 5, Section 5.10.1). If possible, the programmer should avoid loops with iteration counts that cannot be determined at compile time. If the iteration count is not specified, the compiler has no information upon which to base its cost analysis of eligible nested loops. Array elements should be accessed using the minimum stride. A large stride inhibits cache utilization by causing cache lines to be fetched that likely will not be used. And lastly *5)* vector storage references should be avoided within inner loops.

[5] On the 3090, n is the length of the machine's integer function pipeline.

Appendix E

DLX Architecture

In this final appendix we describe the main features of the DLX architecture and dlxcc compiler used during the simulation tests described in Chapter 6.

E.1 Base Architecture

DLX is a simple scalar architecture intended to illustrate the most common features of a reduced instruction set computer, or RISC [HP90]. DLX has thirty-two 32-bit general-purpose registers (GPRs) R0–R31, and thirty-two floating-point registers, F0–F31. Each floating point register can be used individually to hold a single-precision (32 bit) floating point number or used in even-odd pair to hold a double-precision (62 bit) number. A set of special registers is available for accessing status information, one in particular, the FP status register, is used for both compares and exceptions. All movements to/from a status register must be through a GPR. In addition, there is a branch instruction that tests the comparison bit in the FP status register.

All memory references are through loads or stores between memory and either the GPRs or the FPRs. Accesses involving the GPRs can be to a byte, a halfword, or a word. The FPRs can be loaded and stored with single- or double-precision words. Memory is byte addressable in "big-endian" mode with a 32-bit address. And, all memory accesses must be aligned.

Like instruction operations common to most RISC architectures, DLX operations can be divided into four classes: *1)* loads and stores, *2)* ALU operations, *3)* branches and jumps, and *4)* floating-point operations.

Any GPR or FPR can be loaded or stored, but loading R0 has no effect since this register is made to always hold 0. DLX has only a single addressing mode, base register plus 16-bit signed offset. Halfword and byte loads fill the lower portion of a register while the upper portion is filled with either the sign extension of the loaded value or zeros, depending upon the opcode. A Load High Immediate instruction loads the top half of a register while setting the lower half to zero, thus allowing the formation of a full 32-bit constant, with two instructions.

The DLX architecture has a complete set of ALU instructions, including logical operations and shift operations. All of these instructions are either register to register or immediate. In addition, there are move instructions to copy data between GPRs and FPRs and data-conversion instructions to convert data between single- and double-precision. Comparison instructions perform a logical comparison of the values in two register and place the result into a third register, 1 if true and 0 if false. Because comparison instructions "set" a register they are called set equal, set less than, and so forth. There are also immediate forms of these instructions.

Execution control is achieved with four jump instructions and several branches based upon the usual logical conditions: equal, less than, etc. Two of the jump instructions determine their destination address using a 26-bit offset added to the program counter, the third is a plain jump, and the fourth is a jump and link which is used for procedure calls to place a return address in R31. For branches, the condition is specified by testing a register source to test for zero or nonzero, a value that is often the result of a previous compare. Branch target addresses are specified using a 16-bit signed offset that is added to the program counter.

Floating point instructions manipulate floating point registers and indicate the type of precision. Single precision operations can use any of the registers, while double-precision operations use only an even-odd pair designated by the number of the even register. Floating point load and store instructions move data between the floating point registers and memory both in single and double precision, whereas move operations copy floating point registers to other registers of the same type. There are operations to move data between a single floating point register and an integer register, but moving a double precision value to two integer registers requires two instructions. Integer multiply and divide operations work on 32-bit registers, as do conversions between integer and floating point. Floating point compares set a bit in the special floating point status register, and this bit is tested by the branch operations: Branch Floating Point True and Branch Floating Point False.

E.2 The dlxcc C Compiler

Assembly language code for the DLX simulator is created using a special version of GNU C compiler (version 1.37), called dlxcc, outfitted with a back end for DLX [Sta90]. The particular version of dlxcc that was used is one that is under continuing development as part of the McGill Compiler/Architecture Testbed (McCat) [HGMS91]. In addition to being readily available, the GNU C compiler was selected for McCat because of its versatile intermediate form, Register Transfer Language (RTL). In general, RTL is amenable for most common optimizing transformations such as those described by Aho, Sethi, and Ullman, and extensible to other more recent optimizations [ASU86].

The structure of dlxcc consists of two components: *1)* the front end, which generates RTL from program source code, and *2)* the back end, which produces a corresponding assembly program. In the description that follows we mention only the basic features of each, along with a few recent transformation extensions that have been applied as a part of the McCat project.

RTL code, which is almost in one to one correspondence with the assembly code, is not well suited to high level analysis, such as array, loop, and alias analysis, because much data structure information is lost by the time the code reaches the RTL level. In the original GCC compiler, parsing is invoked once to parse the entire source program, and RTL code for each function is generated as the function is being parsed, statement at a time. In the process, a separate syntax tree is created for each statement, the tree, converted to RTL, and the storage, reclaimed. Since the compiler processes a function at a time, no inter procedural optimization is performed.

To correct the above shortfall, the front end was modified to create a complete Abstract Sytax Tree (AST). In the original GNU compiler, tree nodes for expressions were allocated in a temporary stack and freed as each statement was parsed. In the modified compiler, the stack routines were modified to retain the space used by these nodes. The parser was also modified to build the AST for the complete program, and lastly, new nodes to construct different kinds of loop structures, namely, WHILE-loop, FOR loop, and DO loop statements were added. In the modified compiler, optimizing transformations are then performed on these structures before the RTL code is generated. Among the more advanced features that have been recently added are loop unrolling and an inter-procedural alias analyzer.

The back end of the the dlxcc compiler contains most of the common optimizations one might expect to see in a modern compiler, with a few exceptions. Among the optimizations performed by the compiler are jump optimization, common subexpression elimination, constant propagation, strength reduction, loop invariant code removal, and use of machine idioms. The compiler also does induction variable elimination, and it also does both local and global register allocation. However, it does not do delayed-branch scheduling. Among the recent extensions to the GNU backend are several alternative instruction schedulers: Shieh Papahristou, Muchnick Gibbons, and Bernsein's "level" scheduler [SP89, GM86, Ber89]. As noted previously, a Shieh Papahristou scheduler, described in Chapter 5, was used for the experiments described in Chapter 6 (see Section 5.10.2, page 93).

Bibliography

[AC71] Frances E. Allen and John Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, Englewood Cliffs, NJ, 1971. Courant Computer Science Symposium 5, March 29–30, 1971.

[AH82] M. Auslander and M. Hopkins. An overview of the PL.8 compiler. *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, 17(6):22–31, June 1982.

[AK87] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9:491–542, 1987.

[All83] John R. Allen. *Dependence Analysis for Subscripted Variables and its Application to Program Transformation*. PhD thesis, Rice University, 1983.

[AN88] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, June 1988.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Co., 1986.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, MA, 1988.

[Ban90] U. Banerjee. Unimodular transformations of double loops. In *Proceedings of the Third Workshop on Programming Languages and Compilers for*

Parallel Computing, Irvine, CA, August 1990. Also published in *Monographs in Parallel and Distributed Computing*, Pages 192-219, Pitman, 1991

[Bel90] R. Bell. IBM RISC System/6000 performance tuning for numerically intensive FORTRAN and C programs. Technical Report GG24-3611-00, IBM International Technical Support Center, Poughkeepsie, NY, August 1990

[Ber89] D. Bernstein. An improved approximation algorithm for scheduling pipelined machines. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 430-433, January 1989

[BM76] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. North-Holland, New York, 1976

[BW90] H. B. Bakoglu and T. Whitside. RISC System/6000 hardware overview. In Mamata Misra, editor, *IBM RISC System/6000 Technology*, pages 151-161. International Business Machines Corporation, 1990. Order No. SA23-2619

[CAC+81] G. J. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages 6*, pages 47-57, January 1981.

[CCK90] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990. White Plains, NY.

[CGLT89] Robert Cohn, Thomas Gross, Monica Lam, and P. S. Tseng. Architecture and compiler tradeoffs for a long instruction word microprocessor. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 2-14. ACM, April 1989

[Cof76] E. G. Coffman. *Computer and Job-Shop Scheduling Theory*. John Wiley and Sons, New York, 1976.

[Deo74] Narsingh Deo. *Graph Theory with Applications to Engineering and Computer Science.* Automatic Computation. Prentice Hall, Englewood Cliffs, NJ, 1974

[DH79] J.J. Dongarra and A.R. Hinds. Unrolling loops in FORTRAN. *Software-Practice and Experience*, 9:219–226, 1979.

[DHB89] James C. Dehnert, Peter Y.-T. Hsu, and Joseph P. Bratt. Overlapped loop support in the Cydra 5. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 26–38. ACM, April 1989.

[Dij59] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerishe Mathematik*, 1:269–271, 1959.

[Don87] Jack J. Dongarra. The LINPACK benchmark: An explanation. In C. D. Polychronopoulos, editor, *Proceedings of the 1987 Conference on Supercomputing, Athens, Greece*, pages 457–474, Berlin, June 1987. Springer Verlag, LNCS-297.

[Don88] Jack J. Dongarra. Performance of various computers using standard linear equations software in a Fortran environment. Technical report, Argonne National Laboratory, Argonne, IL, February 1988.

[DS90] Robert B. K. Dewar and Matthew Smosna. *Microprocessors: A Programmer's View.* McGraw-Hill Publishing Co., New York, 1990.

[Gao86] G. R. Gao. A pipelined code mapping scheme for static dataflow computers. Technical Report TR-371, Laboratory for Computer Science, MIT, 1986.

[Gao90] G. R. Gao. *A Code Mapping Scheme for Dataflow Software Pipelining.* Kluwer Academic Publishers, Boston, December 1990.

[GM86] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the ACM Symposium on Compiler Construction*, pages 11–16, Palo Alto, CA, June 1986.

[GO90] R. D. Groves and R. Oehler. RISC system/6000 processor architecture. In *IBM RISC System/6000 Technology*. International Business Machines Corp., 1990.

[GOST92] Guang R. Gao, Russell Olsen, Vivek Sarkar, and Radhika Thekkath. Collective loop fusion for array contraction. ACAPS Technical Memo 41, School of Computer Science, McGill University, March 1992.

[GRW86] D. H. Gibson, D. W. Rain, and H. F. Walsh. Engineering and scientific processing on the IBM 3090. *IBM Systems Journal*, 25(1):36–50, 1986.

[Har69] Frank Harary. *Graph Theory*. Addison-Wesley Publishing Co., 1969.

[HGMS91] Laurie Hendren, Guang Gao, Chandrika Mukerji, and Bhama Sridharan. Introducing McCAT – The McGill Compiler-Architecture Testbed. Technical Report ACAPS Memo 27 (in preparation), McGill University, 1991.

[HM90] Larry B. Hostetler and Brian Mirtich. *DLXsim – a Simulator for DLX*. University of California, Berkeley, CA, December 5 1990.

[HP90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[HS84] Mark D. Hill and Alan Jay Smith. Experimental evaluation of on-chip microprocessor cache memories. *Proceedings of the 11th International Symposium on Computer Architecture*, pages 158–166, June 1984. Ann Arbor, MI.

[IBM83] IBM Corporation. *VS FORTRAN Application Programming: Guide, Release 3.0*, fourth edition, March 1983. Order No. SC26-3985-4.

[IBM87] IBM Corporation. *IBM System/370 Extended Architecture: Principles of Operation*, second edition, January 1987. Order No. SA22-7085-1.

[IBM90] IBM Corporation. *IBM RISC System/6000 CD-ROM Hypertext Information Base Library*, second edition, November 1990.

[Kri90] S. M. Krishnamurthy. A brief survey of papers on scheduling for pipelined processors. *SIGPLAN Notices*, 25(7):97–106, 1990.

[KRP+81] D. J. Kuck, Kuhn R., D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the Eighth ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[KW88] S. R. Kleiman and D. Williams. SunOS on SPARC. *Sun Technology*, 1(3):64-77, 1988. Summer.

[Lam90] Monica S. Lam. Instruction scheduling for superscalar architectures. *Annual Review of Computer Science*, 4:173-201, 1990.

[Law76] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Saunders College Publishing, Ft Worth, TX, 1976.

[LS87] B. Liu and N. Strother. Peak vector performance from VS Fortran. Technical Report RC 12849 (#57786), IBM Thomas J. Watson Research Center, Hawthorne, Yorktown Heights, NY, June 1987.

[McG87] McGill University, Montreal, CA. *MUS C/SP User's Reference Guide, Release 1.2*, 1987. Order No. SH20-6921-2.

[McG88] McGill Computing Center, Montreal. *VS FORTRAN*, October 1988. handout.

[Mis90] Mamata Misra, editor. *IBM RISC System/6000 Technology*. International Business Machines Corporation, first edition, 1990. Order No. SA23-2619.

[Muc88] Steven S. Muchnick. Optimizing compilers for SPARC. *Sun Technology*, 1(3):64-77, 1988. Summer.

[Muk91] Chandrika Mukerji. Instruction scheduling at the RTL level. Technical Report ACAPS Note 28, McGill University, 1991.

[Nic88] Alexandru Nicolau. Loop quantization: A generalized loop unwinding technique. *Journal of Parallel and Distributed Computing*, 5:569-586, 1988.

[OHM+90] Kevin O'Brien, Bill Hay, Joanne Minish, Hartmann Schaffer, Bob Schloss, Arvin Shepherd, and Mathew Zaleski. Advanced compiler technology for the RISC System/6000 architecture. In Mamata Misra, editor, *IBM RISC System/6000 Technology*, pages 154-161. International Business Machines Corporation, 1990. Order No. SA23-2619.

[PMSB88] Andris Padegs, Brian B. Moore, Ronald M. Smith, and Werner Buchholz. The IBM System/370 vector architecture: Design considerations. *IEEE Transactions on Computers*, 37(5):509-520, May 1988.

[Pol88] Constantine D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, Norwell, MA, 1988.

[PS82] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

[PW86] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[Rad82] G. Radin. The 801 minicomputer. In *Proceedings of the SIGARCH/SIGPLAN Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 39–47, Palo Alto, CA, March 1982. ACM.

[RG81] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Workshop on Microprogramming*, pages 183–198, 1981.

[Sar89] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Pitman, London and The MIT Press, Cambridge, MA, 1989. In the series, Research Monographs in Parallel and Distributed Computing. This monograph is a revised version of the Author's Ph.D. dissertation published as Technical Report CSL-TR-87-328, Stanford University, April 1987.

[Sar90a] Vivek Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. Technical report, IBM Research, October 1990. Submitted to IBM Journal of Research and Development.

[Sar90b] Vivek Sarkar. The PTRAN parallel programming system. In B. Szymanski, editor, *Parallel Functional Programming Languages and Environments.* McGraw-Hill Series in Supercomputing and Parallel Processing, 1990.

[SC90] Vivek Sarkar and David Cann. POSC: a partitioning and optimizing SISAL compiler. *Proceedings of the ACM 1990 International Conference on Supercomputing*, pages 148–163, June 1990. Amsterdam, the Netherlands.

[SG91] Vivek Sarkar and Guang R. Gao. Optimization of array accesses by collective loop transformations. *Proceedings of the 1991 ACM International Conference on Supercomputing*, pages 191–205, June 1991.

[SP89] J. J. Shieh and C. A. Papachristou. On reordering instruction streams for pipelined computers. *SIGMICRO Newsletter*, 20(3):199–206, August 1989.

[Sta90] R. M. Stallman. Using and porting the GNU CC. Technical report, Free Software Foundation, Cambridge, MA, 1990.

[Tar83] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[Tou84] R. F. Touzeau. A FORTRAN compiler for the FPS-164 scientific computer. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 48–57, June 1984.

[Tuc86] S. G. Tucker. The IBM 3090 system: An overview. *IBM Systems Journal*, 25(1):4–19, 1986.

[UBF+84] David Ungar, Ricki Blau, P. Foley, A. Dain Samples, and David Patterson. Architecture of SOAR: Smalltalk on a RISC. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 355–362, June 1984. Ann Arbor, MI.

[WL90] Michael E. Wolf and Monica S. Lam. Maximizing parallelism via loop parallelism. *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, August 1990. Also published in *Monograph in Parallel and Distributed Computing*, Pages 243–259. Pitman, 1991.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 26–28 1991.

[Wol89] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman, London and MIT Press, Cambridge, MA, 1989. In the series, Research Monographs in Parallel and Distributed Computing. Revised version of the author's Ph.D. dissertation, Published as Technical Report UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign, 1982.

[WS87] Shlomo Weiss and James E. Smith. A study of scalar compilation techniques for pipelined supercomputers. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 105–109. ACM, 1987.

[ZC90] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, 1990.