Parallel Loop Transformation Techniques For Vector-Based Multiprocessor Systems

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Viet Ngo

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

May 1994

Contents

Abstract

Dedicated to my wife Thuy, my son Khoi-Viet

and our parents

Bich Ngo, Mao Mai and Huyen Nguyen, Yen Bui.

Acknowledgements

There are many people without whose help I would not have been able to reach this goal. I would like to thank my thesis advisor, Professor Wei-Tek Tsai, for being patient with me throughout the years. His guidance and insightful suggestions made this work possible. His amazing working ethic has a profound impact on my view toward the research world.

The compiler study group whose members include Sourav Bhattacharya and Gary Elsesser provided invaluable assistance for this thesis. Gary is both an excellent mathematician and programmer, a rare combination. Sourav is an extremely prolific researcher. Their spontaneous insights in many technical discussion sessions or screaming/yelling sessions aided many aspects of this dissertation.

Many thanks go to the people at Cray Research, in particular the pretty darn good compiling system(PDGCS) group. Terry Greyzck provided the vector mapping phase for the outer loop vectorization concept. Clay Andreasen reviewed the design of array privatization and proofread the manuscript. Steve Collins also helped proofread the manuscript. Many thanks to the original VPR team members including Doug Jackson, Jeff Nicholson and Clay Andreasen for building the software platform upon which this research was based.

I am particularly thankful to my brother, Dr. Noel, for helping me with the writing and editing of this dissertation. He also inspired the DIM approach to private array analysis and the theorems in the conditional percolation section.

I would like to thank my parents and my in-laws for having so much faith that I could overcome all the hurdles set by the PhD program. Their wisdom has been my inspiration. Special thanks to my in-laws for providing the support (and foods) so I could do the research.

Last but not least, I would like to give a very special thank you to my wife, Thuy, for her unstinting love, much needed moral support and encouragement to overcome the thesis tar-pit syndrome.

Abstract

A parallel vector processor(PVP) supercomputer relies on parallel processing and pipelines to deliver its enormous computational rate. Loop parallelization and vectorization together enable the performance of application programs to approach the theoretical peak rate of a PVP. This dissertation presents a series of powerful compiling techniques that broaden the classes of loops that can be vectorized and parallelized.

A serial loop can be executed in parallel by different processors if it does not have any loop-carried dependencies(LCD). We propose analysis and transformation techniques to detect and/or remove non-inherent LCD which include: geometric dependence testing methods to perform private array analysis, a code generation method to enable DOALL transformation of loops involving conditionally defined privatized variables, a symbolic analysis technique which in the presence of control constructs allows a compiler to remove false dependencies from irrelevant flow paths.

Loop vectorization has been found to be one of the most significant forms of parallelism. However, it has traditionally been limited to innermost loops and outer loops which are made innermost by the application of loop distribution and interchange. We propose a framework for direct vectorization of outer loops(OLV); i.e., vectorization of an outer loop without interchange or distribution. The framework includes: general vector execution modeling, legality of OLV, vector loop selection and an OLV vector code generator.

These techniques have been implemented in the Cray Fortran-90 compiling system. Implementation details are presented. The efficacy of these techniques are demonstrated by significant performance improvement of application programs.

Chapter 1

Introduction

Parallel processing has emerged as a technology of choice over the last few decades. Computational demand in high performance computing exceeds the *flops* rate available from any single processor system. Fundamental physical restrictions (e.g., speed of light) put an upper bound on the performance that could ever be available from single processor systems. This problem of attaining a higher flops rate than what a single processor can provide has traditionally been approached using multiple processor systems, where a number of processors work together to solve computationally intensive tasks.

The two major sources contributing to the power of supercomputers are hardware technology and multilevel architecture concurrency. Their architectures vary widely; they span the range from vector and array computers which can handle an entire array with a single instruction, to parallel systems with thousands of processors, each of which is capable of executing an independent instruction stream. Any given class of supercomputers is specialized in the sense that it may perform extremely well on one set of applications and very poorly on another. For example, a vector computer may operate at near peak performance when applied in a regular fashion to uniformly structured data, whereas it may deal rather inefficiently with list-processing problems or database management.

An important problem faced by each supercomputer system is the conversion of the large body of existing code, millions of lines of Fortran, into a form suitable for concurrent processing. Supercompilers, which perform automatic restructuring, have been developed to perform this task. They analyze sequential programs, in particular their loops, to detect hidden concurrency, and use this information to create concurrent programs. Thus automatic restructuring relieves users of a supercomputer from the burden of manually transforming old 'dusty desk' programs to run on a new machine.

In general, the complexity of the relationship between algorithm structure, computer architecture and transformation strategies prevents optimal global solutions. Supercompilers cannot transform every program efficiently. More specifically, supercompilers can detect hidden concurrency in sequential algorithms but the current state of the art compiling techniques are not able to construct a new algorithm that is better suited to a parallel architecture. Moreover, while there are many advances toward the principle problems of parallelization, the variety and structural complexity of parallel systems has so far prevented a general solution of the parallelization problem. Furthermore, there is no global compiling technology to detect and extract hidden concurrency in a sequential program and restructure the program for parallel execution.

While many advancements in the compiling technology for multiprocessor systems with vector capability have been made in the past decade, the performance, in million floating point operation per second(MFLOPS), of many time-critical loop nests from important applications is only a fraction of the potential rate deliverable by the underlying hardware. More specifically, the performance of these loop nests can be improved dramatically by recoding the nests in machine language without using additional knowledge over what a compiler can gather. In other words, the hand coding version can expose hidden concurrency that current parallel compiling technology is unable to discover. There are numerous compiling technologies missing which attribute to the poor performance of these loop nests. The most notable weaknesses are due to the compiler's inability to detect concurrency or restructure loops for parallel execution. Hence, the focus of this dissertation is to develop new compiling techniques enabling a supercompiler to better analyze and restructure loops for parallel execution.

1.1 Parallel Architecture

There are many different types of parallel architectures and can be classified into two classes: Single Instruction Multiple Data(SIMD) Multiple Instruction Multiple Data(MIMD). SIMD processors are a synchronous system where the control unit and all computational elements are driven by a common clock. Each computational element acts on different data. In other words, each computational element executes the same code on a different part of the data. A SIMD machine can be thought of as a vector processor. MIMD processors are composed of a set of independent and autonomous processors that are fully or partially interconnected in some way. In MIMD computers, the control is distributed, with each processor having its own control unit and/or local memory. Global control units may be used to coordinate the entire system or to allow for the design of a hierarchical MIMD system. A MIMD machine is programmed as a sequential(or vector) computer.

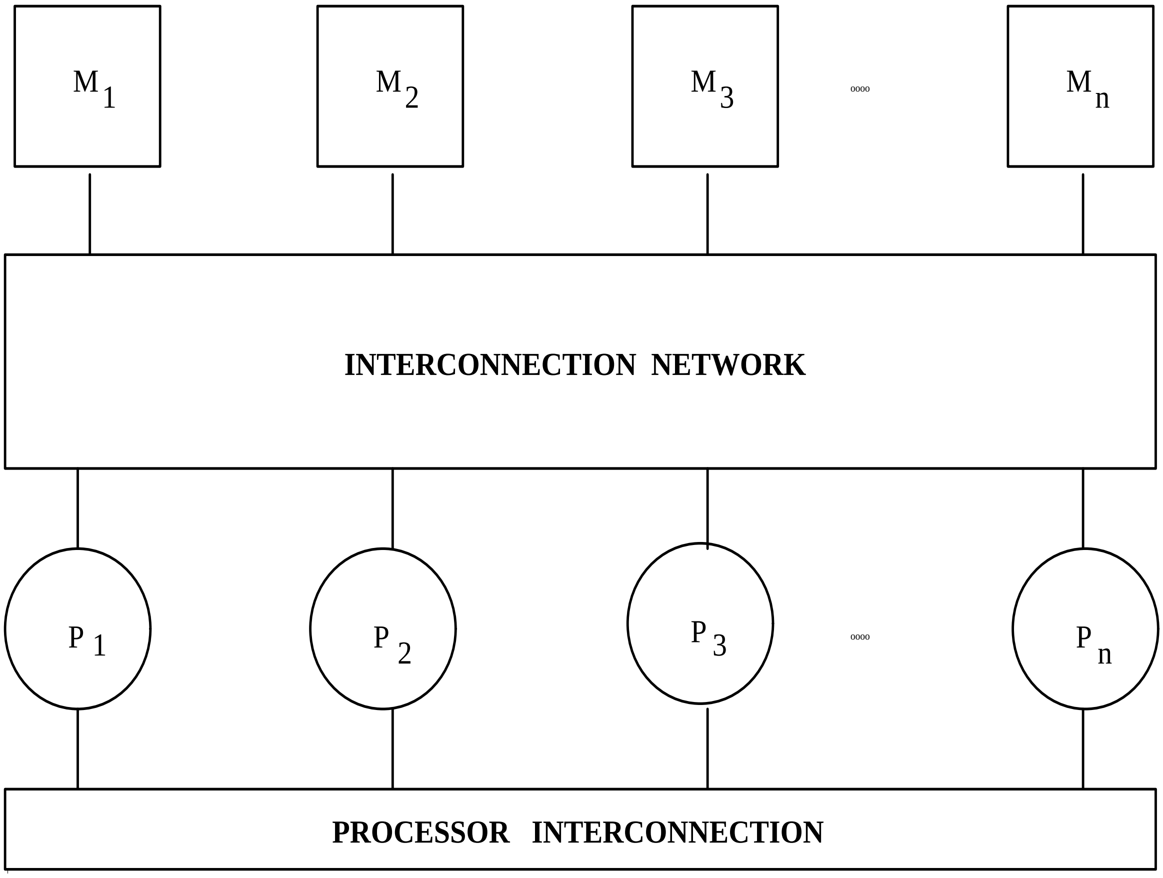

Figure 1.1: A shared memory parallel processor model.

The architecture model used throughout this thesis is a shared memory parallel processor system as shown in Figure 1.1. The machine consists of n processors(numbered 1, 2, ..., n) that are connected to a shared memory M through a multistage interconnection network. The memory can be interleaved and each processor can access any memory module, or can communicate with any other processor through the memory. Each processor has its own private memory that can be organized as a cache, register file or RAM. Each processor is a stand-alone unit. It has its own control unit and can operate independently and asynchronously from the other processors. Our machine model therefore is MIMD. We also assume that each of the processor is a vector or array processor and thus it can operate in single instruction multiple data or SIMD mode. Our model includes machines such as Cray-YMP8, Cray-C90, IBM-3090, NEC SX-3, and Convex-3080, etc.

1.2 Compiler Overview

The principle components of the compiling system being considered in this research are shown in Figure 1.2.

The front-end translates the source language into an intermediate form. This translation may be accomplished in one or more passes over the code, depending on the structure of the source language. Compile-time error checking is usually performed at this stage. This phase is language dependent and mostly/ideally machine independent.

The middle-end consists of several phases, each performing specific transformations on the intermediate form. This phase includes control flow analysis, data dependence analysis, restructuring, loop level parallelization which includes DOALL transformation and vectorization, and classical optimizations. Classical optimizations include strength reduction, loop-invariant code motion, and common subexpression elimination. The middle- end is intended to be both language and machine independent.

The back-end translates the intermediate form into a machine specific form, usually object code. This translation, also called code generation, may require several passes, including: instruction selection, instruction scheduling, and register allocation. The back-end is largely language independent and machine dependent.

This research extends compiling technology in the data dependence analysis phase and the loop restructuring for parallel execution phase of the middle-end. In the middle-end, control flow analysis determines the flow of control of the input program; data dependence analysis examines subscripted variables individually to determine data dependence relation between references of the same array; the restructuring phase transforms the sequential program into vector code or parallel code suit-

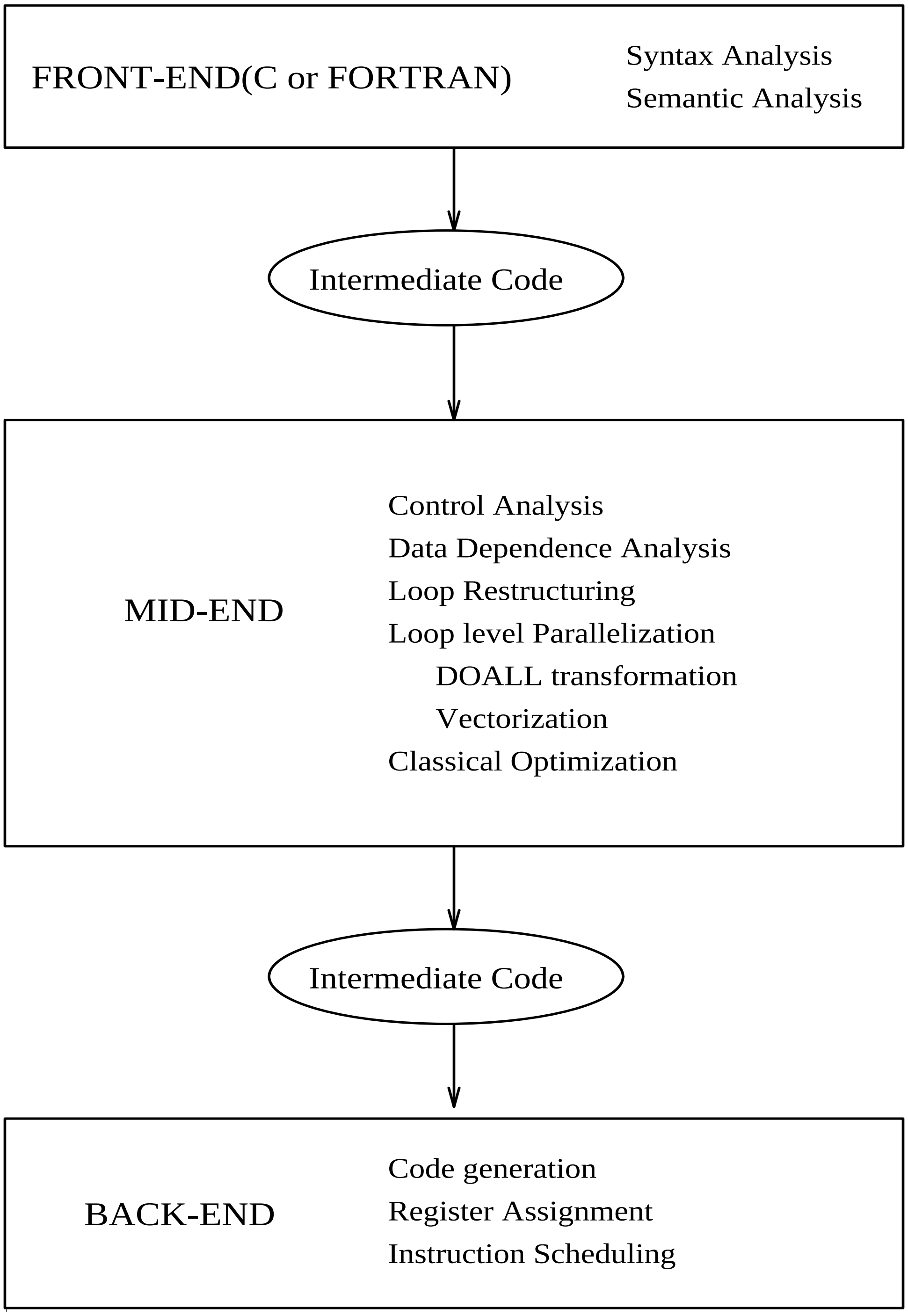

Figure 1.2: Compiler Overview.

able for parallel execution; classical optimizations aim at improving the serial code quality being generated, more details can be found in [1].

1.3 Contribution

This dissertation focuses on parallel compiling technologies for parallel vector processors. A series of new pragmatic compiling techniques are developed to further enhance loop parallelization and vectorization, the two major sources of computational power of these machines. These techniques, incorporated in a production compiling system, are application performance driven and compile time efficient. The compile time efficiency is achieved in both running time complexity and software engeneering solution(by circumventing the problem of maintaining current data flow information via recomputing or incrementally updating).

Array privatization removes storage dependencies, induced by these arrays, which increases the feasibility of transforming serial loops into DOALL loops for parallel processing. We restate the array privatization problem as an*array containment* problem and develop two complementary methods for solving it. Delayed Index Mapping(DIM) and Exact Regular section Descriptor(ERD) are geometric methods for representing the portion of an array which is read/written within the context of a loop nest. By delaying the application of the index maps, DIM can describe access patterns generated via coupled subscripts and triangular iteration spaces. However, DIM handles set union poorly. ERD complements DIM by supporting set union over more restricted types of access patterns.

A pruning technique to analyze dependence information in the presence of conditional constructs is presented. This technique enables a compiler to deduce execution path from symbolic conditional statements, whose values are unknown at compile time, which in turn allows the compiler to ignore irrelevant data dependencies. This technique is used to solve the conditional privatization problem, the problem of estimating the write set in the presence of conditional.

One difficulty preventing DOALL transformation of a serial loop is the conditionally defined privatized variable. The problem is that at compile time it is difficult to determine which processor will be responsible for delivering the last value of a privatized variable. We present a generic solution, a dynamic approach which generates run time code to capture the value produced by that particular processor. The dynamic approach is general, however, the efficiency of the run time code generated by this approach can be improved with a certain class of monotonic write patterns which occur frequently in application programs. We refer to this improvement as the static approach.

We propose a new direct method of vectorizing outer loops; i.e., vectorization of an outer loop without interchange or distribution. *Outer Loop Vectorization* (OLV) offers additional vectorization opportunity, increases vector register utilization, and improves data reuse. In this work, we develop a framework which includes vector execution modeling, legality of OLV or when an outer loop is vectorizable - what is the minimum set of of data dependencies which must be considered to minimize dependence cycles that inhibit vectorization, how to select which loop to vectorize when there is more than one choice. We also present the fundamental construct of an innermost-loop vector code generator and show modifications necessary to produce an outer-loop vector code generator.

These techniques have been implemented in the Cray Fortran-90 compiling system. Loops in different application programs have been parallelized and vectorized using the techniques presented in this dissertation. Overall performance improvement nearly equal the number of participating processors was observed for parallelization, and that of up to 1500% for outer loop vectorization.

1.4 Thesis Overview

The remainder of this thesis is organized as follows.

Chapter 2 contains definitions, concepts, terminologies, and results necessary to describe this research.

Chapter 3 contains results in the area of loop parallelization, DOALL transformation. The array privatization problem is addressed. Two complementary data dependence testing methods are proposed. The problem of capturing the last value of a privatized variable is also addressed. Experimental results including privatization hit-rate, speedup and observations are presented.

Chapter 4 contains results in the area of loop vectorization. A framework to directly vectorize an outer loop is proposed. A basic constraint of vector execution ordering, vector legality to determine when an outer loop can be vectorized, and a vector loop selection metric are presented. Experimental results using application programs are reported.

Chapter 5 contains engineering solutions in the implementation of the results in chapter 3 and chapter 4. The conditional privatization problem is studied in detail. Solutions are presented including a pruning method to prune irrelevant execution paths. An overview of an innermost vector code generator and an outer loop vector code generator are presented.

Chapter 6 concludes the thesis and discusses open avenues for possible future research.

Chapter 2

Preliminaries

This dissertation focuses on exploiting existing analysis to perform effective transformations. To understand these optimizations requires the background knowledge on which they are based. We therefore begin with an overview of the analysis required for program parallelization and transformation.

2.1 Lexicographic Ordering

Let $I = (i_1, i_2, \cdots i_m)$ and $J = (j_1, j_2, \cdots j_m)$ denote vectors of m integers, $I, J \in \mathbf{Z}^m$. We make the following definitions

- I is a zero vector if all of its elements are zero.
- The leading element of I is the first nonzero element.
- $lev(I)$ is the position of the leading element. For example, if i_l is the leading element of I then $lev(I) = l$. If I is a zero vector then $lev(I)$ is ∞.
- The leading negative element of I is the first negative element.
- The leading positive element of I is the first positive element.
- I is (lexicographically) positive or negative if its leading element is positive or negative respectively. (The sign of an integer i is denoted by sign(i), where sign(i)=1 if i is positive, sign(i)=-1 if i is negative, and sign(i)=0 if i=0,

- I is a direction vector if each of its components is one of the integers: 1, 0, -1.
- A distance vector if I, J, denoted by $dist(I, J)$, is defined by $dist(I, J) = J - I$.
- A partial relation $I \prec_l J$ is defined as

$$i_1 = j_1, i_2 = j_2, \cdots, i_{l-1} = j_{l-1}, \text{ and } i_l < j_l.$$

2.2 Dependence Analysis

Dependences describe a partial order between statements that must be maintained to preserve the meaning of a program with sequential semantics. A dependence between statement S_1 and S_2, denoted by $S_1 \delta S_2$, indicates that S_1, the *source*, must be executed before S_2, the *sink*. There are two types of dependence: control dependence and data dependence.

2.2.1 Control Dependence

Intuitively, S_2 is control dependent on S_1, denoted by $S_1 \delta^c S_2$, if during execution, S_1 can immediately affect whether S_2 is executed or not. Control dependence formalizes the notion that execution of one vertex in the graph conditionally depends upon the execution of another vertex in the graph. In the absence of data dependencies, two vertices that have identical control dependencies could be executed simultaneously. Thus, control dependence is used to capture the essential independent control structure of a linear sequence of statements. It is formally defined in terms of the postdominance relationship on the control flow graph. Let G_f denotes the control flow graph of a program. The following formal definitions of control dependence can be found in [19, 23]

Definition 2.1 x is *postdominated* by y in G_f if every path from x to the exit mode of G_f contains y.

Definition 2.2 Given two statements $x, y \in G_f$, y is *control dependent* on x if and only if:

1. $\exists$ a non-null path p, $x \to y$, such that y postdominates every node between x and y on p, and
2. y does not postdominate x.

Based on these definitions, a control dependence graph G_cd can be built with the control dependence edge $(x, y)_l$ where l is the label of the first edge on path $x \to y$. Additionally, if G_f is structured, rooted and acyclic, the resulting G_cd is a tree, where structured has its usual meaning as originally formulated by Bohm and Jacopini [10]. If G_f is unstructured, rooted and acyclic, the resulting G_cd is a DAG [19].

2.2.2 Data Dependence

A *data dependence* between two statements S_1 and S_2, indicates that S_1 and S_2 read or write a common memory location in a way that requires their execution order to be preserved. There are four types of data dependence [32]: *true dependencies, anti-dependencies, output dependencies* and *input dependencies.*

If statement S_1 computes a value used by statement S_2, then S_2 is said to have a true dependence, denoted by $S_1 \delta^t S_2$, on S_1. True dependencies are related to definition-use chains of classical data flow analysis [1]. They often are interpreted as a requirement that the value producing statement be executed before the value consuming statement. However, true dependencies do not reveal all execution constraints. If S_1 uses a value stored in memory location M, and S_2, executed after S_1, modifies the value stored at M, then S_2 is said to be anti-dependent, denoted by $S_2 \delta^a S_1$, on S_1. An output dependence occurs between two statements, $S_1 \delta^o S_2$, that modify the same memory location M. If one statement has an output or anti-dependence on another statement, then the former must be executed after the latter to guarantee that all values used and defined are the same as in the original program. Both output and anti-dependencies are generated by reuse of memory and can be broken safely by careful introduction of new variables. Lastly, If statement S_1 uses a value stored in memory location M and statement S_2 executed after S_1 also uses a value stored in that location, then the dependence in called input-dependence, denoted by $S_1 \delta^i S_2$.

2.2.3 Loop-carried and Loop-independent Dependence

Dependence analysis determines which loops in the program may be run safely in parallel. A dependence between iterations of a loop is called *loop-carried* and prevents the iterations of a loop from being executed in parallel [51]. Consider the following loop

$$
\begin{array}{lll}
 & \text{DO } I = 2, 100 & \\
S_1 & \quad A(I) = \ldots & \\
S_2 & \qquad \ldots = A(I) & \\
S_3 & \qquad \ldots = A(I-1) & \\
 & \text{ENDDO} &
\end{array}
$$

The true dependence $S_1 \delta^t S_2$ is called *loop-independent* because it exists regardless of the surrounding loops. Loop independent dependencies, whether data or control, occur within a single iteration of a loop and do not inhibit a loop from running in parallel. For example, if $S_1 \delta^t S_2$ were the only dependence in the loop, the iterations of this loop could be run in parallel, because statements executed on each iteration only affect other statements in the same iteration and not in any other iterations. However, loop-independent dependencies do affect the statement ordering within a loop iteration. Interchanging statement S_1 and S_2 violates the loop-independent dependence, i.e., changes the meaning of the program.

By comparison, the true dependence $S_1 \delta^t S_3$ is *loop-carried* because the source and sink of the dependence occur on different iterations of the loop. S_3 reads the memory location that was written by S_1 on the previous iteration. Loop-carried dependencies inhibit loop iteration from executing in parallel without explicit synchronization; they, however, allow reordering of the statements.

2.2.4 Dependence Testing

Determining the existence of data dependence between array references is more difficult than for scalars, because the subscript expression must be considered. The process of differentiating between two subscripted references in a loop nest is called *dependence testing* ([6] [7] [24] [38] [49]). To illustrate, consider the problem of determining if there exists a dependence from statement S_1 to S_2 in the following nest:

```
        DO I1 = L1, U1
          DO I2 = L2, U2
            ...
            DO Im = Lm, Nm
S1              A(f1(I1,...,Im),...,fn(I1,...,Im) = ...
S2              ... = A(g1(I1,...,Im),...,gn(I1,...,Im))
            ENDDO
            ...
          ENDDO
        ENDDO
```

Let α and β be vectors of m integers induced within the ranges of the upper and lower bounds of the m loops. There is a dependence from S_1 to S_2 if and only if there exist α and β such that α is lexicographically less than or equal to β and the following system of *dependence equations* is satisfied:

$$f_k(\alpha) = g_k(\beta) \quad \forall k, 1 \leq k \leq n$$

We use δ^* as a short form for δ^f, δ^a, δ^o or δ^i. Suppose that S and T denote statements in a loop body. A dependence T δ^* S is called *backward dependence* iff S lexically precedes T. Otherwise, the dependence is called a *forward dependence.*

2.2.5 Distance and Direction Vectors

Distance and direction vectors may be used to characterize data dependencies by their access patterns between loop iterations. If there exists a data dependence for $\alpha = (\alpha_1, \ldots, \alpha_m)$ and $\beta = (\beta_1, \ldots, \beta_m)$, then the *distance vector* $D = (d_1, \ldots, d_m)$ is defined by $\beta - \alpha$. A direction vector θ of α, β, denoted by $dir(\alpha, \beta)$ is the sign of the distance vector D. Thus for all loop levels, $(1 \leq k \leq m)$, θ is defined as follows:

$$\theta_k = \begin{cases} 1 & \text{if } d_k > 0 \\ 0 & \text{if } d_k = 0 \\ -1 & \text{if } d_k < 0 \end{cases}$$

Dependence distance and direction are represented as a vector elements which, displayed from left to right, represent the dependence from the outermost to the innermost loop in the nest. By definition all distance and direction vectors are lexicographically positive. For example, consider the following loop nest

```
DO I = 1,N
   DO J = 1,M
      DO K = 2,100
         A(I+1,J,K-1) = A(I,J,K) + 10
      ENDDO
   ENDDO
ENDDO
```

The distance and direction vectors for the true dependence between the definition and use of array A are (1, 0, -1) and (<,=,>), respectively. Since several different values of α and β may satisfy the dependence equations, a set of distance and direction vectors may be needed to completely describe the dependence arising between a pair of array references.

Distance vectors, first used by Kuck and Muraoka [31], specify the number of loop iterations between two accesses to the same memory location. Direction vectors, introduced by Wolfe [48], summarize distance vectors and are therefore less precise. However, there are situations where direction vectors may be computed, but distance vectors cannot be.

Both may be used to calculate loop-carried dependencies. Additionally, direction vectors are sufficient to determine the safety and profitability of loop interchange [50] [51]. Distance vectors are often required by other transformations that exploit parallelism ([7] [28] [35] [47] [50]) and improve data locality [16] [17] . Data dependence also characterizes reuse of individual memory locations [16].

2.3 Program Dependence Graph(PDG)

A program dependence graph [19, 23] encodes both control and data dependence information using directed edges to connect the vertices. The vertices in a PDG may represent several concepts: data operators, branch points in control flow, or regions of common control dependence. Edges represent any of several varieties of data dependence or control dependence. Auxiliary edges may be present to indicate special kind of data flow, such as different kinds of procedure parameter passing. A PDG is rooted at a distinct vertex name *Entry*. Every node in a PDG is reachable from Entry via some combination of data or control dependence edges. The PDG construction process uses the control flow graph as a basic building block.

Definition 2.3 (Control flow graph) A control flow graph(CFG) is a directed graph whose vertices include the basic blocks of a program along with two distinguished vertices Entry and Exit. There is a path from Entry to any basic block at which the program can be entered, and there is an edge to Exit from any block that can exit the program. To ensure that the control dependence graph is rooted at Entry, there is an edge from Entry to Exit.

A PDG is constructed by layering two different subgraphs: a data dependence subgraph(DDG) and a control dependence subgraph(CDG). The data dependence subgraph consists of all edges that indicate data dependencies together with all of the vertices that are either sources or targets of data dependence edges. More formally, a PDG is defined as

Definition 2.4 Let $P = < V, E >$ be a program dependence graph. The data dependence subgraph of P is the subgraph $DDG_P = < V', E' >$ where

$E' = \{e \in E \mid e \text{ denotes a data dependence } \}$ and

$V' = \{u, v \in V \mid < u, v > \ \in E'\}$.

The control dependence subgraph includes all of the edges that indicate control dependence, together with the vertices that are the sources (not the targets) of such edges. The root vertex Entry is also included in the CDG.

Definition 2.5 Let $P = < V, E >$ be a program dependence graph. The control dependence subgraph of P is the subgraph $CDG_P = < V', E' >$ where

$$E' = \{e \in E \mid e \text{ denotes a control dependence } \} \text{ and}$$

$$V' = \{u, v \in V \mid < u, v > \; \in E'\} \; \bigcup \; \{Entry\}$$

2.4 Parallel Code Generation

A *parallel loop* is a loop whose iterations are executed in parallel by different processors. There are two types of parallel loops: *DOALL* loops and *DOACROSS* loops. Iterations of a DOALL loop must be completely independent, while the processors executing the iterations of a DOACROSS loop must be synchronized. Thus a DO loop can be transformed into a DOALL loop iff it does not contain any loop-carried dependence. In contrast, every DO loop can be transformed into a DOACROSS loop if all dependencies are satisfied by proper synchronization between loop iterations. In other words, a serial loop can always be transformed into a DOACROSS, while the classes of loops that can be transformed into DOALL are more restricted.

For each program with DOALL loops, a single-program-multiple-data(SPMD) [18] parallel program is generated. A parallel SPMD program consists of sequences of sections of type: *serial*, *parallel region* or *parallel section*. A serial section is executed by exactly one processor. The first processor to arrive at a serial section is assigned to it, while other processors skip the section. The iterations of the DOALL loop which constitutes a parallel section are executed in parallel without synchronization. Iterations are dynamically assigned to processors when they reach the section. After a processor has completed its iteration, it returns to compete for another one. Any processor that reaches the parallel section after all iterations have been executed skips the section and proceeds to the next section. A parallel region is a section of program which is executed by all processors. It could, in principle, be executed by a single processor. A parallel region is used to set up local context for each processor provided that the time to set up local context is much smaller than the synchronization overhead of serialization. In general, one main processor is responsible for executing the program. Additional processors are requested by this processor when a parallel section region is encountered. The additional processors requests can be scheduled to improve load balancing [20] .

2.5 Transformations to Enhance Parallelism

Dependence breaking transformations are used to satisfy specific dependencies that inhibit parallelism. They may introduce new storage to eliminate storage-related anti or output dependencies, or convert loop-carried dependencies to loop-independent dependencies, enabling the safe application of other transformation. If all the dependencies carried on a loop are eliminated, the loop may then be run in parallel.

- **Statement Reordering** interchanges the textual positions of two statements within a loop body. Thus backward dependencies can be converted into forward dependencies by rearranging the order of the statements inside the loop. [51] shows that such reordering preserves loop-carried dependencies; as long as there are no cycles in the dependence graph, statement reordering can be used to convert all backward dependencies into forward dependencies.

- **Loop Distribution**. Statements that can be vectorized or parallelized can be separated from those that cannot by splitting the loop into two or more loops with the same loop header. This transformation is valid if both endpoints of any backward dependencies end up in the same loop. Again, statement reordering can be used to improve the effectiveness of loop distribution. Since a cycle in the dependence graph must include a back edge, the strongly connected regions of the dependence graph form the basic units that cannot be broken up via loop distribution.

- **Scalar Expansion**. Definitions of scalar variables inhibit vectorization by generating single statement loop-carried output and anti-dependencies. If the scalar is used in the loop, it also generates backward dependencies which inhibit loop distribution. A scalar, T, can be expanded into an array by replacing each occurrence of T after the first assignment to T in the loop with a reference to the new array subscripted by the normalized loop control variable.

- **Array renaming**, also known as node splitting [30], is used to break anti dependencies by copying the source of an anti dependence into a newly introduced temporary array and rename the sink to the new array [39, 15]. Loop distribution may then be used to separate the copying statement into a separate loop, allowing both loops to be parallelized.

- **Privatization** makes an array or scalar variable local to a parallel loop, eliminating dependencies on the variable between loop iterations. Array privatization [37] will be covered in depth later in chapter 3.

- **Loop peeling** peels off the first or last k iterations of a loop as specified. It is useful for breaking dependencies which arise on the first or last k iterations of the loop [2].

- **Alignment** moves instances of statements from one iteration to another to break loop-carried dependencies [14].

Chapter 3

Array Privatization

3.1 Introduction

This chapter focuses on identifying cross-iteration dependencies which may be eliminated through *array privatization*. When array A is "privatized" each processing element works with a local copy of A. Let us say that processor P_i uses local array A^i. The elements of A that must be copied into the A^i are called the *copy-in* set for A. We say that A has no copy-in iff the copy-in set of A is empty. Analogously, the *copy-out* set is composed of the elements of the A^i which must be written back to A after the DOALL loop has completed. Figure 3.1 shows where copy-in and copy-out fall into the flow of control. Array covering is a compile time method for verifying that no information is carried between iterations of a candidate loop. Each private array may require copy-in before loop execution and copy-out after. See [12] for a formal description of copy-in, copy-out and privatization.

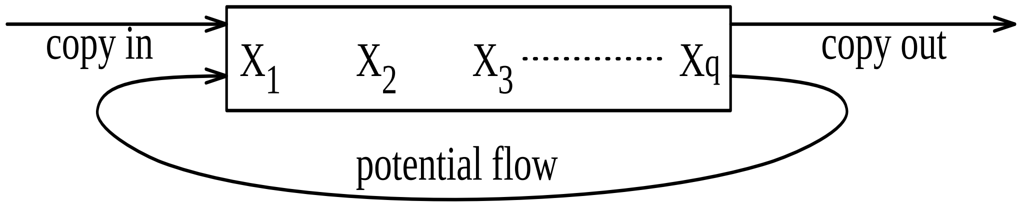

Figure 3.1: Context of Array Privatization

If each array element *read* in a particular loop iteration is *written* in the *same* iteration *and* the write occurs *before* the read, then there is no cross-iteration dependence. This condition is called the *array covering* or *private array* condition. Similarly, if each value read in the loop was generated in the same iteration then no copy-in is required. Each of these conditions may be reduced to a combination of set operations such as union and intersection. The array privatization problem ultimately reduces to array containment and copy-out issues. We propose two approaches to the array containment problem. We handle the copy-out problem in two ways: 1) providing static solutions to a few special cases, 2) developing a dynamic strategy for more general cases.

3.1.1 Related Work

In [12] the privatization problem is decomposed into copy-in, covering and copy-out. Control dependencies are addressed, however, dependencies arising from arrays are not treated. Arrays represent a significant portion of loop-carried dependencies. We focus our attention on the privatizability problem for arrays.

Let R (W) denote the set of elements of an array which are read (written) during one iteration of the loop of interest. Data dependence analysis has been concerned with disproving memory access conflicts; i.e., showing that $R \cap W = \emptyset$. This is because each conflict corresponds to a constraint which must be honored during loop restructuring. This type of data dependence analysis has been extensively studied ([6] [33] [38] [41] [51]).

However, array privatization analysis requires that we test if one array section is contained in another, while the standard dependence analysis methods test if two array sections are disjoint ([3] [6] [15] [40] [49] [51]). Thus we must answer questions of the form $R \subseteq W$ rather than $R \cap W = \emptyset$. Since conventional methods of representing array sections may overestimate write sets (e.g., [1] [4]) it is necessary to revisit this problem. Unlike the empty intersection question, where write sets may be safely overestimated, overestimation of writes sets is not permitted in the containment question. Though it is never incorrect to underestimate the membership of a write set when doing containment analysis, it is beneficial to remain as close to the true write set as possible.

Several schemes have been proposed to address the empty intersection question ([6] [26] [34] [36] [38] [41]). Most permit overestimation of write sets, and are thus unsuitable for array containment analysis. Exhaustive enumeration schemes, such as [36], are useful for interprocedural analysis, but do not address containment analysis. Dependence analysis techniques based on integer programming ([34] [41]) solve the empty intersection problem but do not deal with the array containment question. Geometric models based on convex polytopes have been proposed (e.g., [4] [14] [26]). Since, for most applications array accesses follow simple patterns, these boundaries can often represent W (R) effectively for intersection problems. A significant advantage of these boundary based approaches is their efficient implementation of set operations. Boundary representations, however, may overestimate write sets.

Much of the work on array section descriptors has been motivated by interprocedural data flow analysis (e.g., [4] [5] [13] [14] [15] [26] [46]). Geometric representation of array sections has allowed the effect of array accesses within loops to be effectively summarized. The ability to summarize reference patterns is integral to array covering analysis. But the emphasis in inter-procedural analysis has been on the empty intersection rather than the containment. Since these methods permit overestimation of write sets, they are not directly applicable to array privatization. A single scheme for representing array sections that supports both interprocedural analysis and array privatization is attractive.

The idea of representing read/write array sections geometrically has been proposed by several researchers ([4] [14] [25] [26] [27]). Figure 3.2 (from [26]) compares some of these approaches. It shows how various geometric methods represent the array section indicated by the three small rectangular boxes in 3.2(a). Figure 3.2(a) is the classical summary which treats the entire arrays as a unit [1]. Figure 3.2(b) is proposed in [46]. Figure 3.2(c) is proposed in [36], while figure 3.2(d) shows a bounded regular section with uniform stride ([14] [26]). Figure 3.2(e), which is based on [5], uses simple boundary segments[1].

The space enclosed by the solid line in each diagram indicates the data region represented by the method. Except in figure 3.2(c), these regions are inexact summaries of the accessed regions. The approach in figure 3.2(c) is exact, but it is computationally expensive to perform set operations using this representation for containment analysis. Other approaches contain the data access points, but also contain extra elements. The extra elements (within the boundary but not dashed) form an overestimation of the access set. We refer to these extra elements as the fat-portion in the boundary representation. Similarly, loops with non-unit stride (or non-unit coefficients in subscript expressions) induce gaps in the access region. These gaps comprise the hole-portion of the boundary representation. Both the fat-portion and hole-portion contribute to inexactness.

The array covering problem is closely related to the *array kill* problem studied by [27]. In array kill analysis potential loop-carried flow dependencies ([3]) are broken by intervening updates (kills). Array covering analysis considers both array kill and direct data flow (all values read in an iteration were generated previously in the same iteration). [27] builds on [4] by providing an exactness detection algorithm for array section union and an array kill analysis method based on temporal slices (an enhancement of the *simple section* descriptor of [4]). In [25] data flow analysis is extended to rectangular array sections. Though rectangular array sections are sufficient for many

[1][5] restricts simple segments to the forms $x_i = c$ and $x_i \pm x_j = c$.

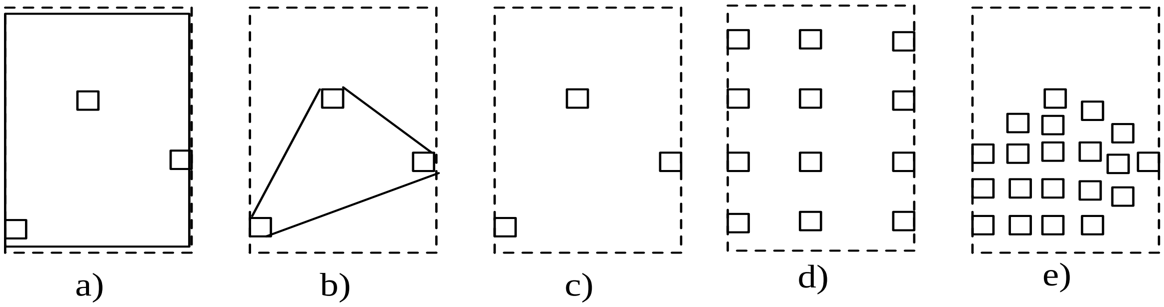

Figure 3.2: Existing Representation Schemes

application domains, the need to address non-rectangular regions remains. [22] provides a general algorithm for array privatization based on parametric integer programming techniques. Compiler experiment is reported in [25], while [27] and [22] do not report experimental results.

3.1.2 Contribution

We present two complementary approaches to the array covering problem: Delayed Index Mapping (DIM) and Exact Regular section Descriptor (ERD). DIM represents an array section by recording an iteration space descriptor and an index map, while ERD represents an array section as a collection of set products. DIM can handle symbolic bounds, coupled subscripts and triangular iteration spaces. However, DIM (generally) must represent a set union as a collection of primitive array sections. ERD complements DIM by supporting union for many cases that appear in practice. ERD can handle conditionals also, using a data flow framework. In addition, we developed static as well as dynamic techniques for the copy-out problem. DIM and ERD approaches in conjunction with the copy-out techniques have been implemented in the UNICOS environment on a Cray Y-MP8 system. They were then evaluated using benchmark application programs (see section 3.6). Experimental observation[2] shows that

1. Most applications (nearly 95 percent of those studied) offer some array privatization opportunity.

2. Among the cases which are potentially privatizable, the DIM and ERD schemes are reasonably ($\approx$ 20 percent) successful. (We present experimental data to demonstrate the success rate of

[2]Refer to section 3.6 for details.

these schemes in section 3.6.)

3. When array privatization occurs, most of the time the loop can be transformed to a DOALL loop. This leads to a direct serial to parallel conversion of the affected loop. In our experiments, a successful array privatization generally leads to DOALL-transformation of the loop.

We compute the speedup rates and make several related observations. Section 3.2 defines the programming model and introduces necessary operations. Section 3.3 develops *DIM* and section 3.4 develops *ERD*. Methods for handling conditionals is included in section 3.4. Section 3.5 examines the *copy-out* problem for several static special cases followed by the development of a dynamic strategy for more general cases. Section 3.6 first describes our experimental environment and then reports results from the experiments. We make several observations from these results. Finally, section 3.7 concludes the paper by summarizing our contribution.

3.2 Program Model and Issues

This section defines our program model and describes how, within our model, the array privatization problem may be reduced to a small collection of operations on array section descriptors. We then present a qualitative comparison of the DIM and ERD approaches.

3.2.1 Model

Figure 3.1 shows context of array privatization. The box represents the body of a potential DOALL loop and the sequence $X_1, \ldots, X_q$ denotes the read/write "stages" of a particular array, call it "A", which is both read and written within the body of a candidate loop. Let W_i (R_i) denote the portion of A which is written (read) during stage X_i. For clarity of presentation, we assume that in each iteration of the candidate loop (each pass through the box) stage X_i is executed completely before stage X_{i+1} begins. Furthermore, we will assume that for each stage, X_i, $W_i = \emptyset \vee R_i = \emptyset$ and both W_i and R_i are loop invariant with respect to the candidate loop.

$$\text{Private iff } \forall j \in \{1, \ldots, q\}, R_j \cap \bar{W}_q \subseteq \bar{W}_j \tag{3.1}$$

$$(R_j \subseteq \bar{W}_j) \vee (\bar{W}_q \subseteq \bar{W}_j) \Rightarrow (R_j \cap \bar{W}_q) \subseteq \bar{W}_j \tag{3.2}$$

$$\text{Array Kill iff } \forall j \in \{1, \ldots, q\}, R_j \neq \emptyset \Rightarrow \bar{W}_q \subseteq \bar{W}_j \tag{3.3}$$

$$\text{Min Copy In } = \bigcup_{j=1}^{q} (R_j - \bar{W}_j) \tag{3.4}$$

$$\text{No Copy-In iff } \forall j \in \{1, \ldots, q\}, R_j \subseteq \bar{W}_j \tag{3.5}$$

Given our program model, we can easily define necessary and sufficient conditions for the privatization of the candidate array, A, in terms of the read sets, R_i, and write sets, W_j. Let $\bar{W}_k = W_1 \cup W_2 \cup \cdots \cup W_k$. A, which is both read and written within the body of our candidate loop, may be made private if and only if each element of A is written before read (equation 3.1). When the computation of intersection proves impractical, we employ (3.2) to avoid the intersection operation. Equation 3.3 expresses the standard "array kill" condition in terms of our read and write sets. From these equations it is clear that the array kill condition is sufficient but not strictly necessary, for array privatization. The minimal safe copy-in set is defined in (3.4). If the difference operation is not available one can still determine when the copy-in set is empty by employing equation 3.5. Thus the array covering and copy-in questions can be reduced to a series of set operations on array section descriptors where containment is essential and union is very important. Intersection and difference of array sections may be avoided by employing weaker testing methods; trading optimization opportunity for implementation cost.

Having reduced the problem to a set operations on array sections, we now focus our attention on the structure of the these array sections. To simplify presentation, we assume that assignments to array elements are embedded in loop nests which contain no internal control flow other than nested DO loops[3]. Since correctness of privatization analysis is not compromised by overestimation of read sets, we place no restriction on the use of conditional logic in the construction of read sets. Figure 3.3 shows the structure of a generic write-read pair. Unless stated otherwise, the functions f_i^j, are assumed to be affine linear transformations with (compile time) constant integer coefficients. Loop bounds are subject to the same restrictions as array indices except that the "constant" term may be a symbolic constant. The strides (s_i^j) must be known at compile time.

The discussion which follows focuses on the implementation of array section operations, particularly containment. Since our representation of array sections is exact and independent of the type of section (read or write), it is possible to use either a read or write section descriptor as either

[3]Our implementation employs a handful of local transformations to recognize common control flow idioms (such as, write to the same location on both sides of a branch) and transform them to the form described above. These transformations enhance DIM and ERD by removing minor obstacles to their application.

```
DO $I_0 = l_0,\ u_0,\ s_0$
    ...
    DO $I_1^i = l_1^i,\ u_1^i,\ s_1^i$
        ...
        DO $I_{n_i}^i = l_{n_i}^i,\ u_{n_i}^i,\ s_{n_i}^i$
            $A(f_1^i(I_1^i, \cdots, I_{n_i}^i), \ldots, f_d^i(I_1^i, \ldots, I_{n_i}^i)) = \cdots$
        ENDDO
        ...
    ENDDO
    ...
    DO $I_1^j = l_1^j,\ u_1^j,\ s_1$
        ...
        DO $I_{n_j}^j = l_{n_j}^j,\ u_{n_j}^j,\ s_{n_j}^j$
            $\cdots = A(f_1^j(I_1^j, \ldots, I_{n_j}^j), \ldots, f_d^j(I_1^j, \ldots, I_{n_j}^j))$
        ENDDO
        ...
    ENDDO
    ...
ENDDO
```

Figure 3.3: A generic Write-Read Pair

operand of any supported operation[4].

3.2.2 Issues

The effectiveness of an array privatization detection scheme, such as DIM or ERD, is determined by its ability to represent array sections and operate on them effectively. Since write sections must be represented exactly, the structure of the defining iteration spaces and subscript expressions is important. When the bounds of an inner loop depend on the induction variables of enclosing loops, the iteration space is called "triangular" [14]. If the induction variable of a loop is incremented by a value other than 1, the loop is said to have non-unit stride. Each of these properties may affects the shape and "density" of and an iteration space, and may cause array covering analysis to fail. Likewise, the relationship between subscript expressions is important. If two or more subscripts of an array reference are functionally dependent the reference is said to have "coupled subscripts". Coupled subscripts, exclude representations that depend on independence of array dimensions. Since triangular iteration spaces and coupled subscripts are not unusual in scientific programming (e.g., LU decomposition and tridiagonal solvers), it is important that a privatization method address these issues.

Given the exact representation of the appropriate array sections, it is necessary that the detection algorithm be able to perform the necessary set operations. Containment is essential (sine qua non) for private array detection. Any representation that does not support containment is inappropriate. Union is next in importance because it is common in scientific programming to initialize complex arrays using a sequence of loops and single assignments. Intersection and difference, through useful, are of secondary significance.

We present two complementary approaches to the array covering problem: DIM and ERD. DIM represents an array section by recording an iteration space descriptor and an index map, while ERD represents an array section as a collection of set products. Table 3.1 summarizes the strengths and weaknesses of each approach. While neither approach can handle all cases, these methods complement each other.

[4] We only require that write sections be represented exactly. It is permissible to overestimate read sections without generating incorrect code.

Capability	DIM	ERD
rectangular iteration space	yes	yes
symbolic loop bounds	some	some
triangular iteration space	yes	no
non-unit stride	yes	yes
coupled subscripts	yes	no
containment ($R \subseteq W$)	yes	yes
union ($W_1 \cup W_2$)	poor	yes
intersection ($R_j \cap \bar{W}_q$)	open	yes
difference ($R_j - \bar{W}_j$)	open	open
conditionals (identical write shape)	yes	yes
conditionals	open	yes

Table 3.1: Benefits of Each Approach

3.3 Delayed Index Map (DIM) Approach

In this section we present the DIM approach to solve the containment question. We start with a preliminary section which contains terminologies and concepts necessary for describing DIM in the subsequent section.

3.3.1 Preliminaries

We consider geometric properties of iteration spaces. A triangular iteration space can be represented by a triangular matrix of the form

$$LT(k) = \{X \in Z^{k \times k} | \ X_{i,i} \geq 1 \ \wedge \ i < j \Rightarrow X_{i,j} = 0\}$$

The range or the address of a read or write access in a triangular iteration space is bounded as shown in the following theorem.

Theorem 3.1 Let $S = \{x \in \Re^{n \times 1} \,|\, L\,x \leq M \ \wedge \ x \geq 0\}$, where $L \in LT(n)$, $M \in Z^{n \times 1}$. S is bounded.

Proof: Define $x^- = \max(0, -x)$.

Since $x_j \geq 0$ we only need to show that $max(x_j) < \infty$ $(1 \leq j \leq n)$.
$L\,x \leq M \Rightarrow L_{j,j}\,x_j \leq M_j + \sum_{i=1}^{j-1} L_j^- \; max(x_i)$.
$L_{j,j} > 0$, so x_j is bounded iff $L_{j,j}\,x_j$ is bounded.
Thus by induction on j, we find $0 \leq x_j \leq max(x_j) < \infty$ (j = 1 .. n).

We extend our study to a larger class, namely the convex iteration spaces. We state some basic definitions.

Definition 3.1 (Convex hull) Let E be a subset of $\Re^{n\times 1}$. Define $H(E)$, the convex hull of E, by the equation:

$$H(E) = \{\sum_{e\in E} \lambda(e)\,e \mid \lambda \in [0,1]^E \wedge \sum_{e\in E} \lambda(e) = 1\}$$

Definition 3.2 (Convexity) We say that a set X in $\Re^{n\times 1}$ is a convex set if and only if $X = H(X)$.

Definition 3.3 (Extreme points) Let S be a bounded convex subset of $\Re^{n\times 1}$. $E(S)$, the extreme points of S, is defined by

$$E(S) \subseteq S \wedge S = H(E(S)) \wedge \forall\, Y \subset E(S), S \neq H(Y)$$

The following lemma shows that the extreme points of bounded convex sets are preserved under linear transformation.

Lemma 3.1 (Extreme Point Preservation) Given the following

- a bounded convex set $S \subseteq \Re^{n\times 1}$
- $A \in \Re^{d\times n}$, $b \in \Re^{d\times 1}$
- $h : \Re^n \rightarrow \Re^d;\;\; h(x) = A\,x + b$

We have: $E(h(S)) \subseteq h(E(S))$.

proof: choose any $y \in h(S)$.

$y \in h(S) \Rightarrow \exists x \in S \; s.t. \; h(x) = y$.
$x \in S \Rightarrow \exists \lambda \in [0,1]^{E(S)} \; s.t. \; \sum_{e\in E(S)} \lambda(e) = 1 \wedge \sum_{e\in E(S)} \lambda(e)\,e = x$.
$y = h(x) = h(\sum_{e\in E(S)} \lambda(e)\,e) = A\,(\sum \lambda(e)\,e) + b = (\sum \lambda(e)\,A\,e) + (\sum \lambda(e))\,b$
$= (\sum \lambda(e)\,A\,e) + (\sum \lambda(e)\,b) = \sum \lambda(e)(A\,e + b) = \sum \lambda(e)\,h(e)$
therefore any $y \in h(S)$ can be written as a convex combination of elements from the set $\{h(e) \mid e \in E(S)\}$, so $E(h(S)) \subseteq h(E(S))$.

3.3.2 Formal Description

An index map is a function which carries induction variables to index values. In the DIM approach the application of the index map is delayed as long as possible so that the convexity of the (normalized) iteration space can be fully exploited. Therefore each array section is represented by a pair: an iteration space and an index map. This section focuses on the DIM containment test. We define the DIM containment test formally and walk though a of small example in which DIM detects privatizability where ERD is not applicable.

Lower Triangular Matrix	$LT(k) = \{X \in Z^{k\times k} \mid X_{i,i} \geq 1 \ \wedge \ i < j \Rightarrow X_{i,j} = 0\}$
Read (Γ_r), Write (Γ_w)	$L_r \in LT(m);\ M_r \in \mathcal{Z}^{m\times 1};$
	$L_w \in LT(n)\ ;\ M_w \in \mathcal{Z}^{n\times 1}$
Normalized Triangular	$\Gamma_r = \{x \in \Re^{m\times 1} \mid L_r\, x \leq M_r \ \wedge \ x \geq 0\};$
	$\hat{\Gamma}_r = \Gamma_r \cap Z^{m\times 1}$
Iteration Spaces	$\Gamma_w = \{x \in \Re^{n\times 1} \mid L_w\, x \leq M_w \ \wedge \ x \geq 0\};$
	$\hat{\Gamma}_w = \Gamma_w \cap Z^{n\times 1}$
Read (f), Write (g)	$A_r \in Z^{d\times m};\ b_r \in Z^{d\times 1};\ A_w \in Z^{d\times n};\ b_w \in Z^{d\times 1}$
Index Maps	$f : \Re^{m\times 1} \rightarrow \Re^{d\times 1};\quad f(x) = A_r\, x + b_r$
	$g : \Re^{n\times 1} \rightarrow \Re^{d\times 1};\quad g(x) = A_w\, x + b_w$
Objective:	$f(\hat{\Gamma}_r) \subseteq g(\hat{\Gamma}_w)$

Table 3.2: Framework for DIM Containment Test

Table 3.2 defines our formulation of the DIM containment test. $\mathcal{Z}$ indicates a symbolic integer constant; i.e., an integer valued expression which is loop invariant with respect to the loop being considered from privatization. We assume that each array reference is enclosed by a looping construct[5] of the form illustrated in figure 3.3. Given this framework, we can constrain each induction variable by an affine linear function of the induction variables of its enclosing loops. Such a system of constraints is referred to as a *triangular iteration space* by Callahan in [14]. While describing DIM, we will assume that all iteration spaces have been normalized so that each induction variable starts at 0 and advances with a step of 1. In principle, any loop nest of the form described by figure 3.3 may be 0:N:1-normalized if the induction variable steps are known at compile time. Γ_w and Γ_r, define

[5]For completeness, references which are not enclosed by any loop may be viewed as if they where enclosed by an appropriate nest of single trip loops.

0:N:1-normalized iteration spaces.

The principal function of the DIM method is to answer read/write and write/write containment questions. For notational convenience, we tag the left operand of "⊆" with "r" ("read") and the right operand with "w" ("write"). Since both operands of the containment test have the same form and mathematical properties, either operand may be a read or a write – irrespective of its nominal tag. Likewise, f and g are simply the index maps associated with the left and right operations of the binary operation "⊆".

Let $E(X)$ denote the extreme vertices of X. The DIM containment test rests on following key observations:

1. Every 0:N:1-normalized iteration space is a bounded convex polytope.

2. For small dimensions (e.g., $d \leq 4$), the number of extreme vertices of a 0:N:1 normalized iteration space is modest.

3. *Vertex preserving property*: If A is a bounded convex polytope and $f : A \rightarrow Y$ is an affine linear map then $f(A)$ is also a bounded convex polytope and $E(f(A)) \subseteq f(E(A))$.

4. If A and B are bounded convex sets then $E(A) \subseteq B \Rightarrow A \subseteq B$.

In section 3.3.1 we defined convexity and showed that a triangular iteration space was bounded. We refer the reader to [8] for background information on linear algebra.

Figure 3.4 gives a concise formal description of the DIM containment test. We call the first predicate the "write map inversion" test and the second the "index map shifting" test. We state the following results.

Theorem 3.2 (Write Map Inversion) Given the framework defined in table 3.2.

$$\exists A_w^{-1} \wedge A_w^{-1} \in Z^{n \times n} \wedge g^{-1}(f(E(\Gamma_r))) \subseteq \Gamma_w \Rightarrow f(\hat{\Gamma}_r) \subseteq g(\hat{\Gamma}_w).$$

proof:

1. $g^{-1}(f(E(\Gamma_r))) \subseteq \Gamma_w \Rightarrow E(g^{-1}(f(\Gamma_r))) \subseteq \Gamma_w$

   Let $h(x) = g^{-1}(f(x))$. By lemma 1, $E(h(\Gamma_r)) \subseteq h(E(\Gamma_r))$.

   Thus, $E(g^{-1}(f(\Gamma_r))) \subseteq g^{-1}(f(E(\Gamma_r))) \subseteq \Gamma_w$

2. $E(g^{-1}(f(\Gamma_r))) \subseteq \Gamma_w \Rightarrow g^{-1}(f(\Gamma_r)) \subseteq \Gamma_w$

   Γ_w is convex, so $E(S) \subseteq \Gamma_w \Rightarrow S \subseteq H(E(S)) \subseteq \Gamma_w$.

   Let $S = g^{-1}(f(\Gamma_r))$.

function $\subseteq ((\Gamma_r, f), (\Gamma_w, g))$: boolean

(* see table 3.2 for definitions. *)

if $\exists A_w^{-1} \in Z^{n\times n} \wedge g^{-1}(f(E(\Gamma_r))) \subseteq \Gamma_w$ then

return True

end if

if $A_w = A_r \wedge \exists x_0 \in Z^{n\times 1}$ $s.t.$ $(A_r\, x_0 = b_r - b_w \wedge E(\Gamma_r) + x_0 \subseteq \Gamma_w)$ then

return True

end if

return $False$

end $\subseteq$

Figure 3.4: The DIM Containment Test

3. $g^{-1}(f(\Gamma_r)) \subseteq \Gamma_w \Rightarrow f(\Gamma_r) \subseteq g(\Gamma_w)$

   choose any $y \in f(\Gamma_r)$.

   $x \in f(\Gamma_r) \wedge g^{-1}(f(\Gamma_r)) \subseteq \Gamma_w \Rightarrow g^{-1}(x) \in \Gamma_w \Rightarrow x \in g(\Gamma_w)$.

   therefore $f(\Gamma_r) \subseteq g(\Gamma_w)$.

4. $f(\Gamma_r) \subseteq g(\Gamma_w) \Rightarrow f(\hat{\Gamma}_r) \subseteq g(\hat{\Gamma}_w)$

   $f(Z^m) \subseteq Z^d$, so $f(\Gamma_r \cap Z^m) \subseteq f(\Gamma_r) \cap Z^d$.

   $\exists A_w^{-1} \in Z^{n\times n} \Rightarrow g^{-1}(Z^d) \subseteq Z^n$.

   choose any $y \in g(\Gamma_w) \cap Z^d$.

   $\exists g^{-1} \Rightarrow \exists x \in \Gamma_w$ $s.t.$ $g(x) = y$.

   but $y \in Z^d \wedge g^{-1}(Z^d) \subseteq Z^n$, so $x \in Z^n \cap \Gamma_w$.

   therefore $g(\Gamma_w) \cap Z^d \subseteq g(\Gamma_w \cap Z^n)$.

   Putting it all together we find:

   $f(\hat{\Gamma}_r) \subseteq f(\Gamma_r) \cap Z^d \subseteq g(\Gamma_w) \cap Z^d \subseteq g(\hat{\Gamma}_w)$.

Theorem 3.3 (Index Map Shifting) Given the framework defined in table 3.2.

$$A_r = A_w = A \quad (\text{so } m = n).$$

$$\exists x_0 \in Z^{n\times 1} \; s.t. \; A\, x_0 = b_r - b_w.$$

We conclude that : $E(\Gamma_r) + x_0 \subseteq \Gamma_w \Rightarrow f(\hat{\Gamma}_r) \subseteq g(\hat{\Gamma}_w)$.

proof: choose any $x \in f(\hat{\Gamma}_r)$.

$x \in f(\hat{\Gamma}_r) \Rightarrow \exists r \in \hat{\Gamma}_r \; s.t. \; x = A\,r + b_r$.

$x = A\,r + b_r = A\,r + (A\,x_0 + b_w) = A\,(r + x_0) + b_w$.

Γ_w is convex and $E(\Gamma_r) + x_0 \subseteq \Gamma_w$, so $r + x_0 \in \Gamma_w$.

but $r, x_0 \in Z^{n\times 1}$, so $r + x_0 \in \Gamma_w \cap Z^{n\times 1} = \hat{\Gamma}_w$.

therefore $x = A\,(r + x_0) + b_w \in g(\hat{\Gamma}_w)$.

so $f(\hat{\Gamma}_r) \subseteq g(\hat{\Gamma}_w)$.

The write map inversion test is illustrated by figures 3.5 and 3.6. As with dependence testing ($R \cap W = \emptyset$), it is permissible to return False when the correct answer is True, but one must never return True when the correct answer is False.

The write inversion map test is conceptually quite simple. If the write map (g) is invertible then each element of the array that is updated by the write loop nest is modified exactly once during the execution of the write loop nest (i.e., $|g^{-1}(x)| = 1, \; \forall x \in \hat{W}$). Given this we can follow each read occurrence from a point in the "read" iteration space ($\gamma_r \in \hat{\Gamma}_r$) to a point in the "read" index domain ($f(\gamma_r) \in \hat{R} = f(\hat{\Gamma}_r)$) to the unique point in the write iteration space whence the element was updated ($g^{-1}(f(\gamma_r))$). The write inversion test works because of the geometry of bounded convex polytopes and the vertex preserving property of affine linear maps. Unfortunately, the obvious dual, a "read map inversion" theorem, does not hold.

Like the write map inversion test, the index map shifting test seeks to find a correspondence between elements in the "read" and "write" iterations spaces. Suppose that $\gamma_1, \gamma_2 \in \hat{\Gamma}_r$. Let $\delta = \gamma_1 - \gamma_2$ and $\Delta = f(\gamma_1) - f(\gamma_2) = f(\delta)$. If $\exists \gamma_3, \gamma_4 \in \hat{\Gamma}_w \; s.t. \; g(\gamma_3 - \gamma_4) = \Delta$ then $f(x + \delta) = f(x) + \Delta = g(x + \gamma_3 - \gamma_4)$. The vector $x_0 = \delta - \gamma_3 + \gamma_4$ is the "shift" between the "read" and "write" iterations spaces. If such a vector exists then we need only verify that $E(\hat{\Gamma}_r + x_0) \subseteq \hat{\Gamma}_w$. The index map shifting test seeks to find a shift (x_0) which will map each extreme point of Γ_r to a corresponding element of Γ_w. Special care must be taken when the shift vector is not unique. If $A_r = A_w \wedge |\{x \in Z^n | A_r\,x = b_r - b_w\}| > 1$ the shift test is becomes a small integer programming problem.

Figures 3.5 and 3.6 illustrate how the DIM containment test, specifically the write map inverse method, works. 3.5(a) shows the original loop and 3.5(b) shows the loop after it has been 0:N:1-normalized. Since normalization does not change the set of elements which are read/written, the result of the containment test is not affected by the normalization process. Having obtained the

```
DO L = 1, MAX
   DO I = 1, N
      DO J = I, 2*I
         A(I, J) = ···
      ENDDO
   ENDDO
   DO I = 2, N
      ··· = A(I, 2*I)
   ENDDO
ENDDO
```

(a)

```
DO L = 1, MAX
   DO I = 0, N-1
      DO J = 0, I+1
         A(I+1, I+J+1) = ···
      ENDDO
   ENDDO
   DO I = 0, N-2
      ··· = A(I+2, 2*I+4)
   ENDDO
ENDDO
```

(b)

Figure 3.5: DIM Example

normal form of the iteration space and index maps, we collect $A_w, b_w, \ldots, M_r$ by inspection. The A and b matrices define the index maps while the L and M matrices describe the iterations spaces. For example, the bottom row of L_w is $[-1, 1]$ and bottom element of M_w is 1 because the upper bound of the J loop in figure 3.5(b) establishes the constraint $J \leq I + 1$ which implies $-I + J \leq 1$. A_w^{-1} exists and is integral[6] so, we proceed to the second condition of the write map inverse test: $g^{-1}(f(E(\Gamma_r))) \subseteq \Gamma_w$.

Since $m = 1$, $E(\Gamma_r)$ is trivial to find. For higher dimensional triangular iteration spaces we exploit the special structure of Γ_r in order to obtain $E(\Gamma_r)$ more quickly[7]. $g^{-1}(x) = A_w^{-1}(x - b_w)$, so computation of $g^{-1}(f(E(\Gamma_r)))$ is now mechanical. In figure 3.6 we have implicitly assumed that $N \geq 2$. If $N < 2$ then $E(\Gamma_r) = \emptyset$ and the containment test returns true because $\emptyset \subseteq \emptyset$.

As demonstrated in figures 3.5 and 3.6, the DIM containment method supports both triangular iteration spaces and coupled subscripts. Unfortunately, it is ill suited for other set operations. In particular, the representation employed by DIM does not support union well; a union of array sections is simply a list of the constituent array section descriptors. Containment is done by testing

[6] $A_w^{-1} \in Z^{n \times n} \Leftrightarrow |\det(A_w)| = 1$; i.e. an integer matrix has an integer inverse iff it is unimodular.

[7] We have developed fast methods for finding the extreme points of triangular iterations spaces of dimension 2 and 3.

$$A_w = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \; b_w = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \; L_w = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \; M_w = \begin{bmatrix} N-1 \\ 1 \end{bmatrix} \; A_w^{-1} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$$

$$A_r = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \qquad b_r = \begin{bmatrix} 2 \\ 4 \end{bmatrix} \; L_r = [1] \qquad M_r = [N-2]$$

$$E(\Gamma_r) = \{\, 0, N-2 \,\} \qquad g^{-1}(f(E(\Gamma_r))) = \left\{ \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \begin{bmatrix} N-1 \\ N \end{bmatrix} \right\}$$

$$L_w \, g^{-1}(f(E(\Gamma_r))) = \left\{ \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \begin{bmatrix} N-1 \\ 1 \end{bmatrix} \right\} \leq \begin{bmatrix} N-1 \\ 1 \end{bmatrix}$$

Figure 3.6: Employing the DIM Method

each member of the descriptor list separately. Equation 3.6 provides the justification for this weak method.

$$(\exists j \in \{\, 1, \ldots, p \,\} \; s.t. \, R \subseteq W_j) \Rightarrow R \subseteq (W_1 \cup W_2 \cup \ldots \cup W_n) \tag{3.6}$$

3.4 Exact Regular Section Descriptor (ERD) Approach

Let $\mathcal{A}$ denote a scheme (representation + algorithms) for manipulating array sections. Due to the diversity of array access patterns, $\mathcal{A}$ ought to be able to handle many special cases. These special cases can be seen *features* of $\mathcal{A}$. Among the many necessary and commonly occurring features the following are notable:

1. Set operations: union and intersection.

2. Coupled subscript.

3. Non-rectangular iteration spaces.

4. Symbolic loop bounds.

5. Access in the presence of conditionals.

Note that DIM can handle features (2) and (3) and some forms of (5), but it does handle feature (1) and it is not clear how DIM can be extended to include (1). Figure 3.7 illustrates how DIM can fail. We consider a nested loop with two writes and a read. Collectively the writes cover the read. However, individually neither write covers the read. Thus, DIM fails to detect array covering where ERD succeeds. Here set union is critical.

```
DO I=1, N
   DO J=1, M
      A(1,J) = ...
      A(2,J) = ...
   ENDDO
   DO K=1, 2
      DO J=1, M
         ... = A(K,J)
      ENDDO
   ENDDO
ENDDO
```

Figure 3.7: Deficiency of DIM approach: an illustrative example.

Having observed that feature (1), was necessary to detect array covering in a significant percentage of our benchmark loops, we designed ERD to complement DIM by supporting union. Thus, our combined approach (a combination of DIM and ERD) addresses each of the features listed above, though neither provides all five features.

In the ERD approach we seek to merge contiguous array accesses into a descriptor. For example, if one access touches A(1, 1..3) and another touches A(1, 4..6) then we can merge these sections into a single descriptor (A(1, 1..6)). Merging may take place in any dimension. This merging process is at the heart of union operation. Though simple, this approach is found effective and covers a large number of applications.

The idea of merging along array access patterns has been proposed by Callahan [14] and Havlak [26] and given the name *Regular Section Descriptor* (RSD). We employ their representation, but use different algorithms due to the *exactness* requirement of array covering analysis. For example, if presented descriptors for A(1, 1..2) and A(1, 4..6) the RSD approach would merge them to form A(1, 1..6). However, due to our *exactness* requirement, we cannot combine these two ranges. In other words, while RSD may cover *holes*, ERD must never introduce additional elements.

Since it is not always possible to merge primitive ERD access descriptors, ERD represents an array section by a collection of primitive access descriptors. With a large number of access patterns, it is computationally expensive to determine a merging sequence which leads to the minimal number of fragments. It is even not clear that minimizing the number of fragments will lead to the best answer to the containment questions.

When ERD fails, the situation is similar to the problem faced by DIM. We must apply some form of equation 3.6 to approximate the set containment predicate. Nonetheless, our experiments have shown that ERD can be effective when compiling "real" programs. Section 3.6 reports the performance evaluation of these schemes. Below we formalize how (subject to our exactness constraint) regular section descriptors are merged.

Let r_j denote the range triple corresponding to a dimension j of a particular primitive array section. Then, r_j has the form $l_j : u_j : s_j$, and r_j represents the index set $\{ l_j + ks_j | k \geq 0 \wedge l_j + ks_j \leq u_j\}$. By construction, $s_j \geq 1$. For a d dimensional array, there would be d such range triples. Collectively, the r_i represent a primitive array access pattern. We assume that each dimension is independent of others, so ERD does not support coupled subscripts or non-rectangular iterations spaces.

Let D_1 denote the r_j of a particular access and D_2 denote those of another access. Let l_i^j (u_i^j, s_i^j) denote the value of l_i (u_i, s_i) for D_j. We first take care of the case where one section is contained in the other. After this we either have one section, the larger of the two inputs, or two basic sections. The basic merging operation is expressed in Fig. 3.8.

ERD provides a natural complement to DIM. DIM is able to represent most array sections and ERD supports most array operations. It would be interesting to develop a combined approach.

3.5 Copy Out

Array copy-out occurs when at least one array element is *live* [1] following the privatized loop. Generally, for array privatization to be practical it is necessary for the compiler to determine the processing element which last (according to sequential evaluation semantics) wrote to the exported array element. In general the processor responsible for delivering a particular data element is hard to spot. This section develops several static special case solutions and a generic dynamic case approach.

find k s.t. $r \neq k \Rightarrow l_r^1 : u_r^1 : s_r^1 = l_r^2 : u_r^2 : s_r^2$

if no such k exists, return "merge fails"

if $s_k^1 = s_k^2$ then

    if $l_k^1 \leq l_k^2 \wedge l_k^2 \leq u_k^1 + s_k^1$ then

        return $l_k^1 : u_k^2 : s_k^1$

    else if $l_k^2 \leq l_k^1 \wedge l_k^1 \leq u_k^2 + s_k^2$ then

        return $l_k^2 : u_k^1 : s_k^1$

    end if

end if

return "merge failed"

Figure 3.8: Merging Algorithm

3.5.1 Static Special Cases

We examine three special cases in which the source processor can easily be identified. These cases occur frequently in real programs. The following discussion assumes the program model described by Section 3.2.

- The model which we present in this paper fits the form illustrated in figures 3.9(a) and 3.10(a). Since the shape of the write sections is independent of the iteration being serviced, all exported array values are owned by the processor which served the last index value.

- In figures 3.9(b) and 3.10(b) the write set increasing with each sequential iteration. Again, the processor which services the final index value will own all exported array elements. It may be possible to automatically identify this important case using an automated form of symbolic induction proof (turned to the array covering problem domain).

- Figures 3.9(c) and 3.10(c) illustrate the least desirable case; each processor owns one or more copy-out elements.

The special-cases illustrated above may be sufficiently typical to justify special treatment. Copy-Out is more difficult to characterize in the presence of multiple writes or non-monotonic access patterns.

DO I = 1, 10	DO I = 1, 10	DO I = 1, 10
DO J = 1, 10	DO J = 1, I	DO J = I, 10
A(J) = $\cdots$	A(J) = $\cdots$	A(J) $\cdots$
ENDDO	ENDDO	ENDDO
ENDDO	ENDDO	ENDDO
$\cdots$ = A(expr)	$\cdots$ = A(expr)	$\cdots$ = A(expr)
(a)	(b)	(c)

Figure 3.9: Example Copy Out Forms

3.5.2 Dynamic Approach

The idea behind copy-out is to retain the last written copy of the concerned variable. Here "last written" implies the latest executed DOALL iteration index value. This is because, the latest executed index value bears the semantic significance of the variable with regard to the serial execution. We propose a general method to achieve this for every private variable. Intuitive idea behind this method is explained below.

Every time a value is written into a private variable (x), we note down the DOALL iteration index (I). Note that each processors owns a private copy x, and thus every write into x is treated as a write into a 2-tuple. First component of this 2-tuple stores the value of x, while the second component updates the value of I. In this update of I, if the previous I-value is lower (higher) than the current i-value, then only we re-write the new I-value for increasing (decreasing) index loops. This approach assumes a monotonic dispatching sequence of the different DOALL loop indices.

When the DOALL loop is completed, each PE reports its own private copy of x along with the latest locally written index value I. Finally, all PEs pass through a critical region to elect the globally latest written value of x. This method is well suited both for scalar and array variables. In the case of array variables, the global latest selection process is done by another DOALL loop (attached to the end of the original DOALL). Different PEs computes the globally latest copy of each individual array element.

Fig.3.11 shows a sample DO loop which can be parallelized if the private variable x (which is conditionally defined) can be successfully captured after the completion of the loop. In other words, after completion of the loop one should get the semantically correct value of x. Since the loop, if

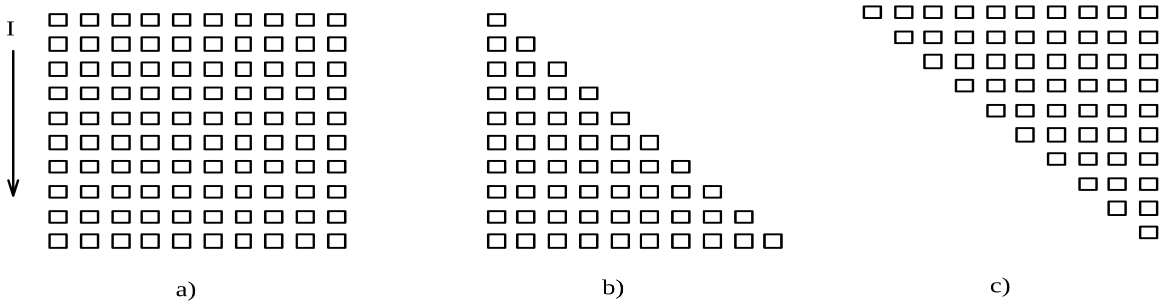

Figure 3.10: Copy Out Patterns

DO $I = 1, N$

IF (...) THEN

$x = ...$

ENDIF

ENDDO

$... = x$

Figure 3.11: Scalar-private: Sample DO loop with conditionally defined private scalar variable x.

parallelized, would span into multiple processors, it is not obvious to identify which processor should carry the eventual *correct* value of x.

Note that in this regard "correct" conforms to the semantics conveyed by the sequential program. Since the sequential program in Fig.3.11 is *increasing* in its loop count - we would finally involve a maximizing operation among loop indices. In the other case, i.e., when the loop count is *decreasing* - a minimizing operation is necessary.

Fig.3.12 shows the parallelized version of the DO loop in Fig.3.11. As explained above, each iteration writing a value into x also writes the corresponding iteration index. At the completion of the DOALL loop, the processors undertake a global maximum operation among themselves to decide which iteration index has written latest a value to x. Note that if the DO loop in Fig.3.11

```
global_I=0
parallel region
    DOALL I=1,N
        If (...) Then
            X.value = ...
            X.index = I
        EndIf
    END DOALL
    guard
        If (global_I > X.index) Then
            x = X.value
            global_I = X.index
        EndIf
    end guard
end parallel region
```

Figure 3.12: Parallelization: Parallelizing the DO loop in the Scalar-private example.

were a decreasing loop, a global minimization operation would have been involved.

A final comment applies to the dispatching sequence (of the DOALL loop iterations). The above code implicitly assumed a monotonically increasing (i.e., in the same sequence trend as the loop index I) dispatching of the iteration index. Thus, the statement "X.index = I" effectively brings out the *local maximum* iteration index value executed by each processor.

However, with random dispatching sequence (or any other dispatching sequence which does not conform to the sequence trend of the loop index) additional sophistication to Fig.3.12 is required. Intuitively, the idea behind generating "X.index" is to capture the *latest writing iteration index value.* Let I_1, I_2, I_3 be three successive (but not necessarily consecutive) iteration index values processed by a particular processor. In an increasing loop (i.e., DO I=1,N) we capture Max(I_1, I_2, I_3), while in a decreasing loop (i.e., DO I=N,1,-1) we capture Min(I_1, I_2, I_3). This can be generalized to an arbitrary number of indices processed by a processor.

Suppose, $I_1 > I_2$ in an increasing loop. When the processor executes the loop body with $I = I_1$, it generates $X.value$ and stores I_1 in "X.index". Subsequently, when the same processor works with $I = I_2$, it generates a new $X.value$. However, since $I_1 > I_2$, the processor cannot delete the value of X for $I = I_1$. It must generate X locally for $I = I_2$, use it and exit - without destroying the old X (for $I = I_1$). This is achieved by keeping two copies for x - one local, and another locally latest. Usage of variable x in a particular loop iteration involves the local copy, while the locally latest copy keeps track of the entire dispatching sequence. The following code augments Fig.3.12 to perform this task.

The augmented code in Fig.3.13 is more complex than the simple version in Fig.3.12. This complex version, though always correct, is more expensive and can be replaced by the simple version. Table3.3 lists the cases where simple version is adequate, and the cases where complex version is required.

	Dispatching Sequence		
Original Loop Index Sequence	increasing	decreasing	random
Increasing	simple version	complex version	complex version
Decreasing	complex version	simple version	complex version

Table 3.3: Choice of code template for handling dynamic copy out.

The above illustration for dynamic copy out handling is with respect to a private scalar variable. For arrays, the same technique can be extended by treating each component of the array. Maximizing

```
....
parallel region
    X.index = 0
    DOALL I=1,N
        IF (...) THEN
            X.local_value = ...
            IF (I > X.index) THEN
                X.index = I
                X.value = X.local_value
            ENDIF
        ENDIF
    END DOALL
end parallel region
....
```

Figure 3.13: Modified version of Parallelization example to handle random dispatching sequence.

(or minimizing) operation for each array element is done using another DOALL. Different processors prepare the "final copy out" elements for different array variables.

3.6 Experimental Results

This section presents experimental results regarding the performance of DIM and ERD. We implemented these algorithms on a Cray $Y-MP8^{TM}$ system in the UNICOS[8] 6.1 environment. Several application benchmarks, including Perfect Club and Linpack, were examined. Our private array analysis methods were added to the existing dependence analysis and program restructuring phase of a production compiler.

Detection of private arrays in many applications significantly improved the overall loop-parallelization rate. In other words, many loops which were forced to run serially before array privatizability was detected can now run as DOALL loops. In this section, we report experimental data which shows: a) the success rate of our private array detection schemes and b) the consequent speed-up due to parallelization.

We use 48 independent programs available at Cray from vendors. The majority of these programs are for scientific applications, however, they also include a few utility applications as well. The size of these programs varied widely, ranging from 30k *LOC (lines of code)* to a few hundred lines. Each program involved several loops, nested and/or simple.

The Cray Y-MP8 provides 8 processors with shared memory [43]. Our experiments were done under the UNICOS operating system using Cray's $Autotasking^{TM}$ [42] and program restructuring tool *PDGCS*. We added our array privatization scheme to *PDGCS* and used *atexpert* [45] to portray the theoretical and actual speed-up factor when parallelization was successful.

3.6.1 Privatization Hit-Rate

In this subsection we answer the following questions:

a) How important is the private array problem? How often does array privatization allow a serial loop to become a DOALL loop?

b) Within our set of application benchmarks, how often does DIM succeed?

c) How often does ERD succeed?

---

[8]UNICOS is a register trademark of Cray Research

We measure success using *hit-rate*. *Hit-rate* is defined as the number of successes in a certain sample. In our experiments the input sample are the application programs. However, an entire program (consisting of several loops and an even larger number of individual read/write pairs) is a large chunk and defining a single success/failure attribute value for the entire program is likely to obscure many internal details. For example, in an application with ten potentially privatizable loops, if at least one loop is successfully privatized then we mark the entire application as a successful event. However, by doing this we lose information above the success rate of other loops. This arises due to the coarse granularity at the application level.

To alleviate this problem, we report *hit-rate* for three distinct granularity levels: application, loop and pairwise-containment. At the application level, if any loop within it is privatized using our scheme then we mark the application as "successful". The next lower granularity is at "loop" level. A nested loop (of arbitrary depth) is counted as a single loop. A particular loop is marked "successful" if at least one particular array in it is privatized. Finally, the lowest level of granularity is at the individual "read-write" (containment) pair. A particular containment pair is marked successful, containment can be proven (using DIM or ERD).

Our input sample space has 48 applications, several thousands of loops, and tens of thousands of containment-pairs. We ran using DIM and ERD and computed *hit-rate* for each level of granularity. We define three metrics for this purpose.

- Metric 1: This metric indicates how important *array privatization* is for real applications. For each application/ loop/ containment-pair we check whether it refers to a private array. If so, a *potential* array privatization is expected. We count the total number of potential array privatization cases. The ratio of this value to the total sample size is called the *hit-rate*. These three ratios are referred to as Metrics 1a, 1b and 1c (for application/ loop/ containment-pair) respectively.

  Metric 1a $= \frac{46}{48}$, Metric 1b $= \frac{436}{1394}$, Metric 1c $= \frac{7848}{11239}$,

- Metric 2: This metric shows how effective the DIM method is. Within the sample space of potentially privatizable arrays (metric 1) we count how many are detected using DIM. Note that, each potentially privatizable array may not be ultimately privatizable due to other constraints (besides the $R \subseteq W$ constraint). In such cases, not having been able to detect the private array is not a deficiency of the DIM scheme. Thus, metric 2 indicates the *worst case* performance of DIM approach. Similar to metrics 1a, 1b and 1c we define metrics 2a, 2b and 2c as the ratio of successful events to the total number of potential array privatization cases for three different levels of granularity: application, loop and containment-pair.

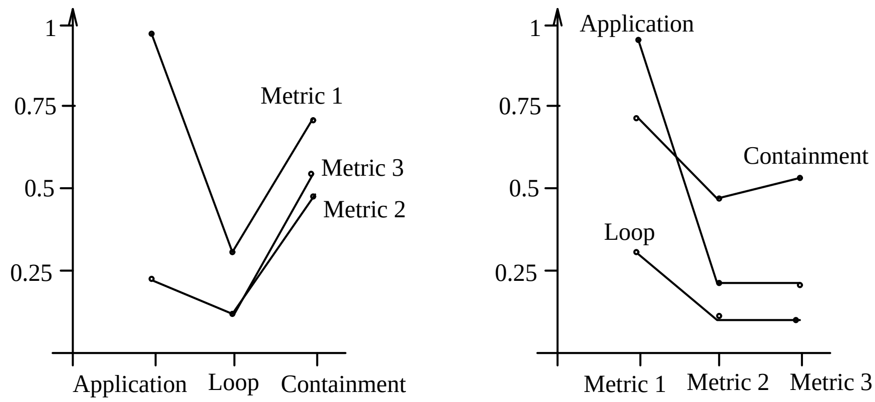

Figure 3.14: Kirk's Diagram: Inter-relationship between granularity and hit-rate.

Metric 2a $= \frac{10}{46}$, Metric 2b $= \frac{53}{436}$, Metric 2c $= \frac{3873}{7848}$,

- Metric 3: This metric is identical to metric 2, however, instead of using DIM we use ERD. Following a similar approach, metrics 3a, 3b and 3c are defined for three different levels of granularity: application, loop and containment-pair.

  Metric 3a $= \frac{10}{46}$, Metric 3b $= \frac{53}{436}$, Metric 3c $= \frac{4304}{7848}$,

The inter-relationships between these mean values are displayed using Kirk's diagram ([29]) in figure 3.14. We make two observations from these diagrams which relate the trend of the mean values and their actual significance.

Observation 1: Array privatization occurs at individual loop level. Marking an entire application as "successful" on account of the success in *any* one of its loops introduces a high skew factor. In other words, hit-ratio metric values for "application" level granularity is likely to be largest. This is reflected in Figure 3.14(a).

On the other hand, satisfying a specific containment-pair may not always lead to successful privatization of the associated array. For example, there may be another containment-pair with the *same* array which does not succeed or there may be other dependencies. This causes the hit-ratio at loop-level much smaller than at containment-pair level.

Observation 2: Metric 1 uses an optimistic estimate when enumerating potentially privatizable arrays, while metric 2 counts only successful privatization events. As a result, metric 1 values are higher than those of metric 2.

However, metric 3 uses ERD which is an improvement over the method used for metric 2. Thus, metric 3 is expected to be *greater than or equal to* metric 2. This is reflected in figure 3.14(b). At the lowest level of granularity, i.e., at containment-pair level, this improvement is prominent. At the other two levels performance remains the same as in metric 2 (i.e., no worse).

Figure 3.15a (3.15c) shows the ratio of the number of loops that were actually detected to the number of loops present in an application, using DIM (ERD). The horizontal axis in these two plots indicate different application programs, while the vertical axis denotes the ratio. Note that the ratio is (by definition) less than one. Similarly, Figure 3.15b (3.15d) shows the ratios of the number of containment-pairs that were actually detected to the number of containment-pairs present in an application, using DIM (ERD).

Observation 3: The hit-rates of DIM (ERD) at loop level are moderate: close to ten percent. However, when a hit occurs, it it likely that a large fraction (if not all) the loops in that application will be transformed. This trend is different from another possible trend, in which DIM (ERD) would have been successful for about ten percent of the loops of each application. This non-uniformity of loop level hit-rates is evident in figures 3.15a (3.15c).

Observation 4: The horizontal axis in Figure 3.15a (3.15c) [as well as Figure 3.15b and 3.15d] places application programs from similar vendors (of similar categories) close to each other. Also, the hit-rate shown in these figures shows a burst pattern. This implies that the hit-rate is application, programmer or programming style dependent. For certain classes of programs the hit-rate is quite high, while for certain other classes it is low. Since, all these programs are scientific applications, we conclude that hit-rate is more likely to depend on programming style than on application type.

Applications which reported high hit-rate are (a) 3D Euler equations solved with implicit algorithm central differences and full geometry, (b) Transonic Navier-Stokes Program, (c) Analysis of transonic flow past a wing by solution of unsteady Euler equation etc. Applications which reported low hit-rate are (a) Circuit equation solver, and (b) Simulator for 2-dimensional fluid flow on a grid.

Observation 5: At containment analysis level we also observe the characteristics similar to those mentioned Observation 3. However the non-uniformity of hit-rates is not as prominent here. There are several instances with medium range hit rate (i.e., around 0.5). This is because at the individual containment pair level each pair has moderate likelihood to succeed. However, for an array to be privatized, all containment pairs relating to this array must succeed. This introduces contributes to the bursty nature in Figure 3.15a and Figure 3.15c.

Observation 6: The hit rates of DIM and ERD do not show any difference at loop granularity level. This is because the effect of improved containment pair detection gets masked out when an entire loop is considered - similar to the masking effect explained in Observation 5. However, the

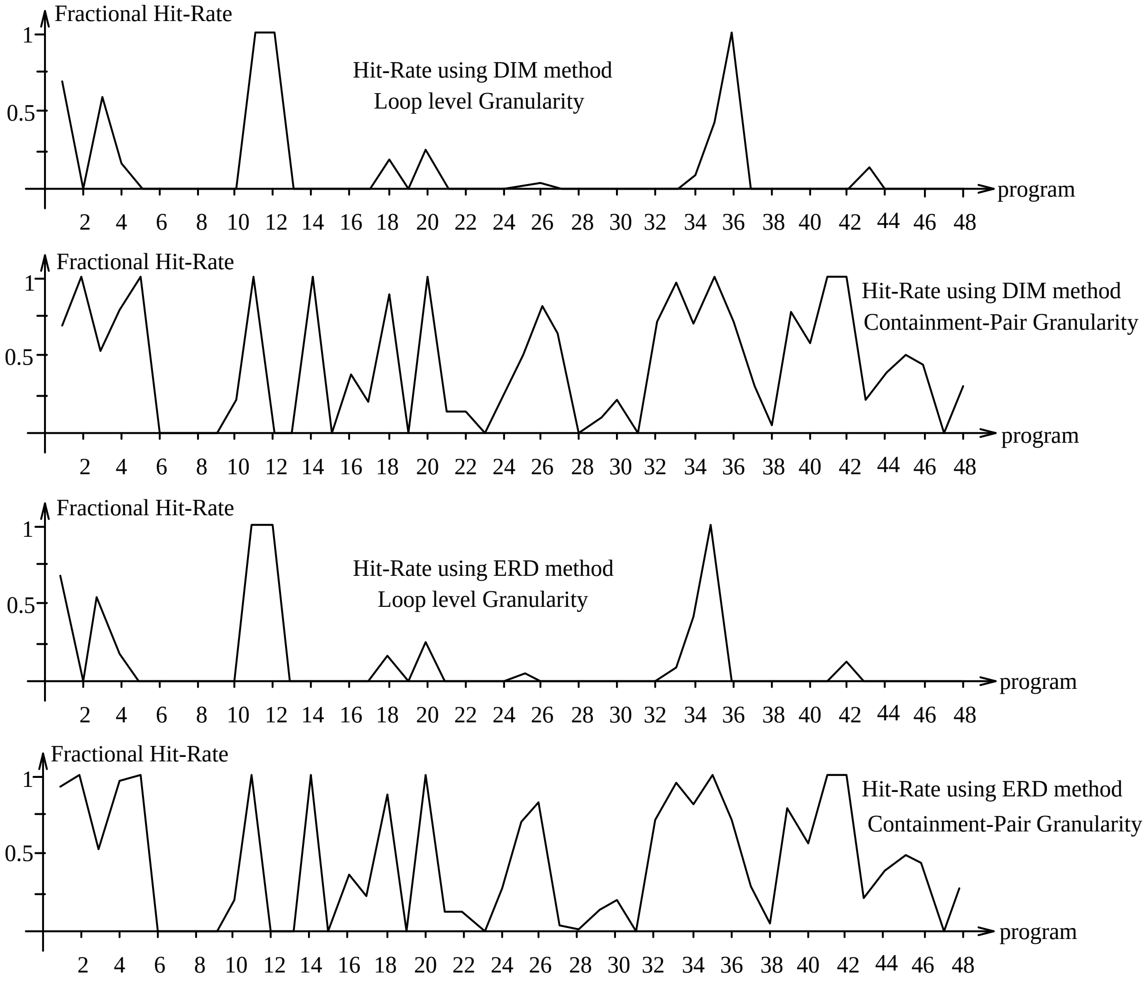

Figure 3.15: Hit-rate plots: DIM and ERD schemes at loop and containment pair levels.

improved performance of ERD can be noticed at the containment pair granularity level.

3.6.2 Speedup Performance

When an array can be privatized (and assuming no other loop-carried dependence exists), the iterations of the loop can be executed in *parallel*. So DIM and ERD may enable transformation of many serial loops into *DOALL*-type. Our implementation in the Cray UNICOS environment demonstrates this potential. In this subsection we present a sample experimental result regarding the speedup. We use atexpert to estimate theoretical and actual speedups for 2 – 8 processors. Figure 3.16 shows a sample screen-dump of the result obtained with one nested loop.

3.7 Summary

When there is no *loop-carried dependence* (LCD) [40], a serial DO loop can be transformed into a parallel (DOALL) loop thereby decreasing its execution time. In application programs LCD can originate from two types of sources: (1) those inherent to the problem domain and (2) those due to storage-dependence among locally computed arrays. The former inhibits DOALL transformation of the loop and (generally) cannot be overcome. However, in the latter case, DOALL transformation may still be possible by privatizing the locally computed array to each participating processor.

To privatize a locally computed array, three issues must be addressed: *copy-in*, *covering* and *copy-out*. Array-containment analysis focuses on the copy-in and covering problems. Most existing dependence tests permit overestimation of write sets because they are designed to solve the empty intersection problem. Since array containment analysis does not permit overestimation of write sets, most existing dependence tests are unsuitable for containment analysis. Existing methods for addressing array kill and data flow issues do not support triangular loop bounds and non-unit index stride. Array privatization methods based on integer programming techniques have been proposed, but experimental results using these methods have not been reported.

Our focus is to develop a practical solution to the array containment problem. We define the *array-containment* problem and develop two methods (DIM and ERD) to address it. DIM supports triangular iteration spaces, coupled subscripts and symbolic loop bounds while ERD provides union and intersection for a more restricted class of array sections. Both methods support array section containment.

We also address the *copy-out* problem. The idea behind copy-out is to retain the written copy of the concerned variable. This is problematic since at compile time the compiler does not know

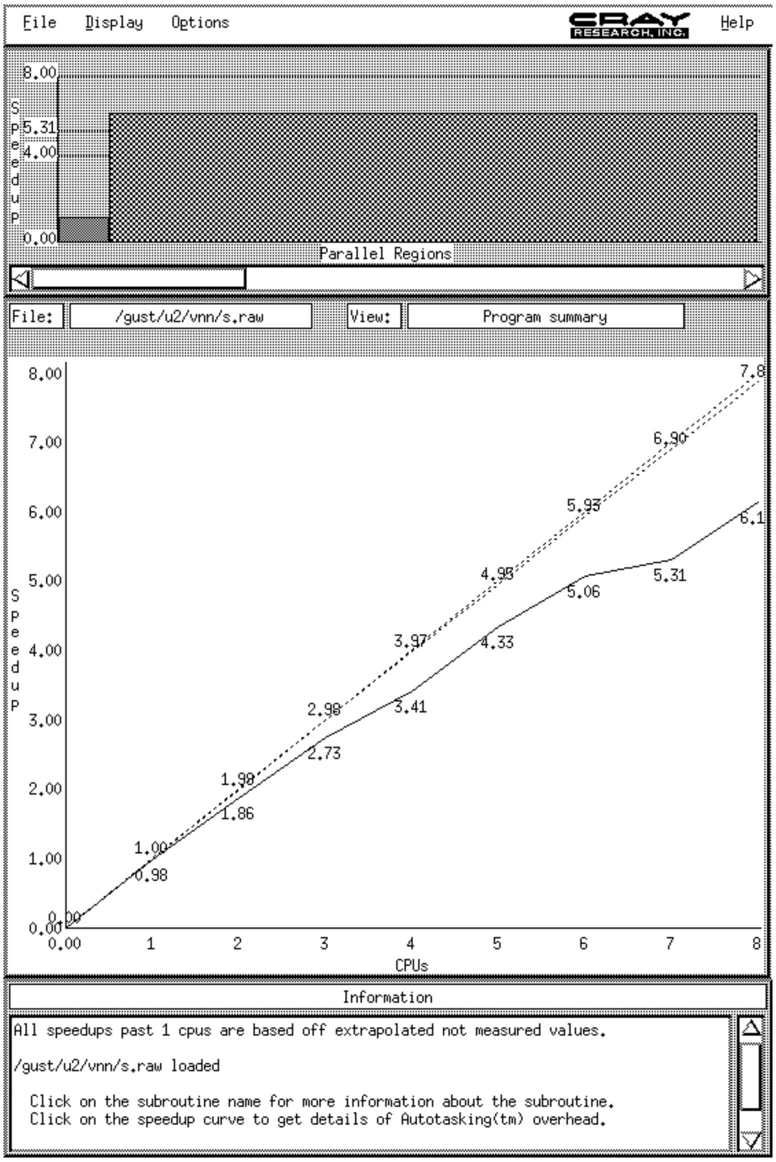

Figure 3.16: Sample screen-dump from our experimental environment.

which processor has the last written value, especially when the variable is conditionally defined. We present a generic solution, a dynamic approach which generates run time code to capture the last written value of a privatized variable. The dynamic approach is general, however, the efficiency of the run time code generated by this approach can be improved with certain class of monotonic write patterns which occur frequently in application programs. We refer to this improvement as the static approach.

DIM and ERD are implemented in the UNICOS environment on a Cray Y-MP8 system. Experimental results lead to several interesting observations: (1) array privatization has a significant impact on loop-parallelization, (2) together, DIM and ERD can recognize a large percentage of array privatization opportunities. (3) In cases where privatization is successful, loops are transformed to DOALL - it leads to a significant speedup.

Chapter 4

Outer Loop Vectorization

4.1 Introduction

This chapter focuses on developing techniques to contribute more to the overall degree of vectorization. Vectorization is one of the most fundamental and effective technique for exploiting parallelism. A vector processor contains a set of pipelines which can operate in parallel and may allow chaining [43]. Furthermore, scalar operations can often be overlapped with vector computations. For example, the loop

```
DO I = 1, 64
   A(I) = A(I) + 5
ENDDO
```

can be vectorized by fetching 64 different A(I) values to a vector register, add 5 to each of the vector's element and store the resulting vector back into A(I). In contrast, serial execution will perform 64 fetching, 64 addition and 64 storing operations. The total vector operations for the above loop are {fetch,add,store}, while serial execution requires 64×{fetch,add,store} scalar operations.

4.1.1 Outer Loop Vectorization: Motivation and Traditional Approach

With nested loops, vectorization leads to more issues. The simplest approach to vectorization in a nested loop is to vectorize the innermost loop. However, there may be better opportunities in vectorizing non-innermost loops. The traditional approach to such cases is to interchange the positions of the innermost loop and the target non-innermost loop, resulting in the innermost placement of the target loop which can then be vectorized. However, such loop interchange may not always be legal; even when interchange is legal, interchanging can potentially destroy vector reuse opportunity which could result in performance degradation. We illustrate these issues using the example in Figure 4.1.

```
(1)  DO I = 1,64
(2)      IF A(I) = 0 THEN
(3)          T = B(I)
(4)          DO J = 1,12
(5)              T = T/C(I,J)
(6)          ENDDO
(7)          B(I) = T
(8)      ENDIF
(9)  ENDDO
```

Figure 4.1: Outer-Loop vectorization in a two level nest

The naive vectorization technique would suggest vectorizing the J-loop, which however has a trip count of only 12 and cannot yield much performance improvement. The other alternative is to vectorize the I-loop. The traditional practice is to interchange the I and J loop positions, bring the I-loop as the innermost and then vectorize the I-loop.

In order to carry out the loop interchange operation, it is necessary to convert the above into a perfect nest. Lines 2, 3 and 7 hinder the formation of a perfect nest. Line 2, i.e., the IF-conditional can be pushed inside the J-loop. But, lines 2 and 3, which have been generated by scalarization techniques[17], cannot easily be eliminated. There is no known compiler technique which can reverse-scalarize back; forward scalarization is popular since it yields several advantages, but reverse scalarization has not been developed. Thus, the above loop example cannot be transformed into a perfect nest, which implies that the I-loop cannot be made innermost and hence the I-loop cannot be vectorized. In other words, the traditional approach fails to vectorize the outer loop.

Even if the loop-interchange were to be successful, e.g., if lines 3 and 7 were absent, there may be more than one choice of loop for vectorization. Suppose, the above loop, instead of being 2-level nested, was 3-level nested with an additional loop DO K=1,64 at the outermost level. Now, both the J-loop and K-loop are vectorization targets. The following questions arise:

- Under what conditions a particular loop may or may not be vectorizable ?
- Or, are all the loops always vectorizable (vector legality) ?
- Which loop should eventually be vectorized (vector selection) ?
- And, finally if we do decide to vectorize a particular loop, what level of performance improvement can be expected ?

These questions cannot always be answered by the existing loop interchange followed by innermost loop vectorization techniques. A framework is needed for guiding the outer (i.e., non-inner) loop vectorization process, which to our knowledge is lacking at this moment.

4.1.2 Direct Vectorization

Our notion is to carry out vectorization keeping the loop direct, i.e., by not having to interchange the target loop to the innermost position. With respect to the example above, it implies that the I-loop would be vectorized keeping its own (i.e., outer) position. This obviously alleviates the problems and disadvantages associated with loop-interchange.

However, vectorization of a non-innermost loop introduces an additional concern: what are the execution constraints that must be followed or satisfied by such outer loop vectorization (OLV). Besides, the other concerns (such as vector legality, vector selection, and expected performance) must be resolved. We propose a framework to guide the OLV process with respect to these objectives.

4.1.3 Contribution

There are two main contributions in this chapter. First, we propose the idea of direct vectorization, where a non-innermost loop can be vectorized without having to interchange the loops. This allows a compiler to avoid several disadvantages (or often impossibilities) of loop-interchange, apart from the execution time expense of conducting the interchange. Second, we propose a framework towards the OLV process. This framework allows the compiler to identify which loops are vectorizable, and among them which one is the most cost-effective to vectorize. A composite performance metric is proposed, which captures the most significant cost parameters, and can reasonably predict the performance improvement that can be expected if a particular loop is vectorized.

The remainder of this chapter is organized as follows. Section 4.2 presents related loop transformation techniques for vectorization, and reviews previous works on OLV. Section 4.3 develops a basic constraint in vector execution order that must be followed by any OLV techniques. This constraint is then used to develop the framework for vector legality, vector selection and expected performance issues. Section 4.4 presents empirical performance results, i.e., real-code benefits using our proposed framework. Section 4.5 concludes the chapter.

4.2 Program Model and Previous Work

This section presents the related transformations for vectorization, the programming model being used in this chapter and previous work.

4.2.1 Related Loop Transformations

The fundamental transformations involved in vectorizing a loop include statement re-ordering, loop distribution, strip-mining and vector statement generation. These are summarily described as follows.

Statement re-ordering interchanges the textual positions of two statements within a loop body. Loop distribution splits a loop with more than one statement into a sequence of loops by distributing the control over single of less-sized group of statements. Strip-mining blocks the iteration space into strips, where each strip's length is equal to or smaller than the vector length of the underlying vector register. Vector statement generation is essentially mapping the input source code into machine vector code. It involves code generation for vector load and store operations.

In terms of legality, i.e., whether or not a particular loop transformation can be carried out, two statements can be re-ordered if there is no loop-independent dependence between the two statements. Loop distribution is legal if there is no loop-carried dependence pointing backward. Strip-mining a loop is always legal.

4.2.2 Program Model

We consider a n level nested loop shown in Figure 4.2. A level k loop is denoted as L_k where $(1 \leq k \leq n)$. Each $B_{k,l}$ where $(1 \leq k \leq n)$ and $(1 \leq l \leq 2)$ consists of a combination of sequences of assignment and conditional statements. There are $(N_1 \times N_2 \times \cdots \times N_n)$ instances of the innermost loop body $B_{n,1}$. Γ will be used as a generic term to denote the iteration space of L_k, $(1 \leq k \leq n)$. Let S and T be two assignment statements in the loop body, we write $S < T$ if S lexically precedes T. The lexicographical ordering of statements is total. Let $\ll$ denote serial execution order. The serial execution order of the instances of $S \in B_{n,1}$ is defined by $S(I) \ll_l S(J)$ iff $I \prec_l J$ where

$$
\begin{array}{ll}
L_1: & \text{DO } I_1 = 1, N_1 \\
& \quad B_{1,1} \\
& \qquad \ldots \\
L_m: & \qquad \text{DO } I_m = 1, N_m \\
& \qquad\quad B_{m,1} \\
L_{m+1}: & \qquad\quad \text{DO } I_{m+1} = 1, N_{m+1} \\
& \qquad\qquad B_{m+1,1} \\
& \qquad\qquad \ldots \\
L_n: & \qquad\qquad \text{DO } I_n = 1, N_n \\
& \qquad\qquad\quad B_{n,1} \\
& \qquad\qquad \text{ENDDO} \\
& \qquad\qquad \ldots \\
& \qquad\qquad B_{m+1,2} \\
& \qquad\quad \text{ENDDO} \\
& \qquad\quad \ldots \\
L'_{m+1}: & \qquad\quad \text{DO } I'_{m+1} = 1, N'_{m+1} \\
& \qquad\qquad \ldots \\
L'_n: & \qquad\qquad \text{DO } I'_n = 1, N'_n \\
& \qquad\qquad\quad B'_{n,1} \\
& \qquad\qquad \text{ENDDO} \\
& \qquad\qquad \ldots \\
& \qquad\quad \text{ENDDO} \\
& \qquad\quad \ldots \\
& \qquad\quad B_{m,2} \\
& \qquad \text{ENDDO} \\
& \qquad \ldots \\
& \quad B_{1,2} \\
& \text{ENDDO}
\end{array}
$$

Figure 4.2: Generic loop nest

$I, J \in \Gamma$, or can be written as $S(I) \ll_\theta S(J)$ iff $\theta = dir(I, J)$. A similar notation is used to express dependencies. We say that $S(I)\ \delta_\theta\ T(J)$ iff $S(I) \ll_\theta S(J)$.

Let $G = < V, E, T >$ be the associated program dependence graph for the loop model where

- V is the set of all nodes in Figure 4.2 together with START, STOP, MULTI and LOOP nodes as described in section 2.3.
- E is the set of all dependence edges. E is a subset of $V \times V \times Z^n \times L$ where L is the set of dependence types: L= {control, anti, flow, output},
- T is a node type mapping. Given a node $m \in V$, $T(m)$ identifies the type of node m as one of the following: START, STOP, IF, MULTI, LOOP.

Let $e \in E$, then e is a dependence edge of the form $e = (S, T, \theta, \alpha)$, where $S, T \in V$, with $\theta \in Z^n$ and $\alpha \in$ { control, anti, flow, output}. We use $e = (S, T, \theta)$ as a short hand to denote a data dependence edge.

Let $G' = < V, E' >$ denote the data dependence subgraph of G where

$$E' = \{e = (S, T, \theta, \alpha) \in E : \alpha \notin \{control\}\}.$$

We augment G' to form a level data dependence graph. The data dependence graph for a loop at level k is represented by $G_k = < V_k, E_k >$, a k-th level dependence graph of G', where

- $V_k = \{\ u \in V : u \in B_{h,l}$ where $h \geq k$ and $1 \leq l \leq 2\ \}$ and
- $E_k = \{e = (S, T, \theta) \in E'$ and $S, T \in V_k : lev(\theta) = k$ or $lev(\theta) = \infty\}$.

A level k program dependence graph is a graph formed by layering the control and G_k subgraphs. Let $G_{p_k} = < V_{p_k}, E_{p_k} >$ denote a program dependence graph for some loop at level k where

- $E_{p_k} = \{e = (S, T, \theta, \alpha) \in E : e \in E_k$ or $\alpha = \{control\}\}$ and
- $V_{p_k} = \{u, v \in V : \exists \theta, \alpha \ni (u, v, \theta, \alpha) \in E_{p_k}\}$.

A dependence path in G_{p_k} is a path $\pi = P(G_{p_k})$ of the form $\pi = S_1, S_2, \cdots, S_t$ where S_r $(1 \leq r \leq t)$ are assignment statements and for every q $(1 \leq q < t)$ the dependence $S_q \delta^* S_{q+1}$, where $* \in$ { control, anti, flow, output}, holds.

Let $C(S, G_{p_k})$ denote the set of all cyclic paths from S to S in G_{p_k}, where S is a statement in V_k. We have $C(S, G_{p_k}) = \{\pi : \pi = P(G_{p_k}) = S_1, S_2, \cdots, S_t$ is a path in G_{p_k}, where $t \geq 1$ and $S_1 = S_t = S\}$.

4.2.3 Outer-Loop Vectorization: Previous Work

The potential benefits of OLV has been identified and a number of works reported the idea of loop-interchange followed by innermost loop vectorization. Loop interchange [7, 50, 51] is one of the most powerful restructuring transformations. It is being used to achieve different optimization objectives such as vectorization, parallelization and locality. In vectorizing, it is use to improve vectorization by moving loops with cyclic dependencies outward and loops without dependencies inwards.

In [50] vectorizing an outer loop is done using interchange. It mentioned the idea of "Super vector" performance, a technique which moves an innermost vectorizable loop outward to enhance memory reuse. [50] also discusses nested loop vectorization, where the method converts a loop nest into multi-dimensional vector operations.

A framework is provided in [3] [51] to generate vector code for a statement S at all levels by determining the smallest number c_0 such that the serial execution of the c_0 outer most loops suffices to break the dependence cycle; the remaining innermost loops, if any, can then be executed in vector mode. An outer loop is not vectorized if there is a recurrence on any of its inner-loop. Loop interchange is used to move the recurrence outward.

[11] proposes a knowledge-base approach which work with the user interactively to come up with the best possible program to be submitted to the compiler. Outer-loop vectorization was proposed in their framework to improve register reuse. The framework uses loop-interchange to 'innermost' an outer vector candidate loop.

4.3 Outer-Loop Vectorization Framework

This section makes the primary contribution in this chapter, and the performance of these contributions are presented in the next section.

4.3.1 DOVEC: Specifying Vectorization

Two form of loop structures are popular: the serial loop and parallelized version (i.e., DOALL) of it. Neither of them can appropriately handle some of the desired syntax for a loop to be vectorized. Consider a loop

```
DO I=1,2
  S(I)
  T(I)
ENDDO
```

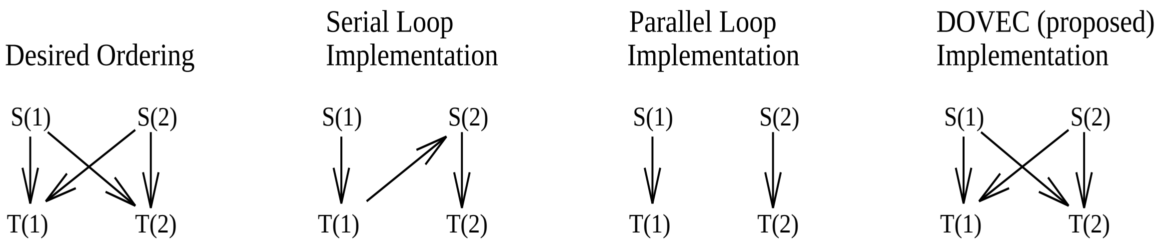

Figure 4.3: The rationale behind DOVEC syntax proposition.

where S and T represent assignment statements. Suppose, there is a forward loop-independent dependence between S(I) and T(I). But, let S(1) and S(2) be parallely executable, and so are T(1) and T(2).

If this is declared as a serial loop then the execution order is

$$S(1) \rightarrow T(1) \rightarrow S(2) \rightarrow T(2),$$

which does not allow any opportunity for vectorization. On the contrary if this is declared as a DOALL loop, then the two threads (S1 → T1) and (S2 → T2) can be executed parallely. Yet, neither of these two lexical structures allowed specifying the other form of vectorization, which is

Step 1: Vector execution of S(1) and S(2)

Step 2: Vector execution of T(1) and T(2)

Note that Step 1 must be totally completed before Step 2 can begin execution. In other words, there ought to be four execution orderings: S(1) → T(1), S(1) → T(2), S(2) → T(1) and S(2) → T(2). The serial loop syntax does not implement the S(2) → T(1) part, despite implements some unwanted orderings such as S(1) → S(2) and T(1) → T(2). The DOALL loop misses out S(1) → T(2) and S(2) → T(1) parts.

Towards this we propose the DOVEC construct which precisely implements the above four (and only four) execution orderings, as shown in Figure 4.3. Syntactically, the DOVEC construct used the word 'DOVEC' instead of 'DO' in the loop header.

4.3.2 Vector Execution Ordering Rules: A Basic Constraint

First, we consider a single DOVEC loop without any inner nesting depth, i.e., a single level loop, with I as the iteration count and two statements S and T constituting the loop body. S lexically precedes T. Let i_1 and j_1 be two instances of the I trip count. Thus, $I, J \in \Gamma$. Let vl denote the degree of parallelism in the hardware vectorizer, e.g., with a 64-element vectorizer vl would be 64. The vector execution ordering constraints (ν^{vl}) of this single loop is given as follows.

1. $S(i_1) < T(j_1)$ *iff* $\lfloor \frac{i_1}{vl} \rfloor \leq \lfloor \frac{j_1}{vl} \rfloor$
2. $T(i_1) < S(j_1)$ *iff* $\lfloor \frac{i_1}{vl} \rfloor < \lfloor \frac{j_1}{vl} \rfloor$
3. $T(i_1) < T(j_1)$ *iff* $\lfloor \frac{i_1}{vl} \rfloor < \lfloor \frac{j_1}{vl} \rfloor$
4. $S(i_1) < S(j_1)$ *iff* $\lfloor \frac{i_1}{vl} \rfloor < \lfloor \frac{j_1}{vl} \rfloor$

For example, with ν^1 (i.e., when $vl = 1$, a serial execution), the above set of conditions become equivalent to what is described in [7]. When vl is very large (i.e., ν^∞), then the condition (1) above is trivially implied, while the condition (2) is not satisfied. Thus, with the generic DOVEC construct, as introduced in Section 4.3.1, where vl can be arbitrarily large - the condition (2) can never be met. However, we would show that in *mixed* nest, as follows, an equivalent set of conditions can be proposed.

A mixed loop nest is a nest of loops consisting of several DO loops and a single DOVEC loop. Let (L_1, L_2, ..., L_{l-1}, L_l, L_{l+1}, ..., L_m) denote a mixed loop nest where the l^{th} loop L_l is a DOVEC loop. Every other loop in the mixed nest is a DO loop. Let $B(I)$ denote an instantiation of the loop body of L_l.

Definition 4.1 (Vector Execution Order: ν_k) The vector execution order at level l for L_l, denoted by ν_l^{vl}, is defined by the following rules (ν_l is a short hand for ν_l^∞):

1. If I and J are index values such that $I \prec_k J$ where $k \neq l$, then all iterations of B(I) shall be executed before the iteration instance B(J).

2. If $I = J$; S and T are statements in B; and $S < T$, then instance $S(I)$ of S shall be executed before the instance $T(I)$ of T.

3. Let I and J be index values such that $I \prec_l J$. Let I$' = (i_1, i_2, \ldots, i_{l-1}, 0, i_{l+1}, \ldots, i_m)$, J$' = (j_1, j_2, \ldots, j_{l-1}, 0, j_{l+1}, \ldots, j_m)$, and statement $S < T$ in B.

   3.1 If $I' = J'$, then $S(I)$ shall be executed before $T(J)$.

   3.2 If $I' \prec_k J'$ where $l < k \leq m$, then the iteration $B(I)$ shall be executed before the iteration $B(J)$.

   3.3 If $J' \prec_k I'$ where $l < k \leq m$, then the iteration $B(J)$ shall be executed before the iteration $B(I)$.

For example consider the mixed loop nest in Figure 4.4. The loop nest is of three levels, of which the innermost and outermost level loops are Do-loops and the middle loop is a DOVEC. The execution orders are shown in Figure 4.5. Figure 4.5a shows the serial execution order, while Figure 4.5b shows the execution order implemented using the DOVEC construct. Figure 4.5c shows two serial ordering edges that are no longer in the vector execution order.

```
DO I_1 = 1,2
    DOVEC I_2 = 1,2
        DO I_3 = 1,2
            S(I_1, I_2, I_3)
            T(I_1, I_2, I_3)
        ENDDO
    ENDDOVEC
ENDDO
```

Figure 4.4: Vector execution order: a small example

4.3.3 Vector Legality

The transformations necessary to perform innermost loop vectorization is well studied [3, 50, 51]. However, the feasibility of vectorizing a non-innermost loop has not been addressed and is the focus of this section. This section will develop a framework to determine when it is feasible to vectorize an outer loop.

Vectorization of a loop involves ensuring vector execution does not violate the data dependence imposed by statements in the loop and rewriting the loop body with a sequence of vector statements. Vectorization of a loop at level k, L_k, requires ensuring that

1. The data dependence relation(generated by the loop) is not violated by the constraints of ν_k.

2. There is no dependence cycle within the statements in the loop and all dependencies are forward.

We focus on condition (1) and derive a condition for which ν_k of all statement instances at L_k is semantically valid. L_k being non-innermost doesn't change the applicability of statement reordering and loop distribution to satisfy condition (2). The following theorem ensures G_k contains the necessary data dependence edges to verify whether or not ν_k violates a program semantic.

Theorem 4.1 The execution order constraints imposed by $E' - E_k$ are satisfied by ν_k.

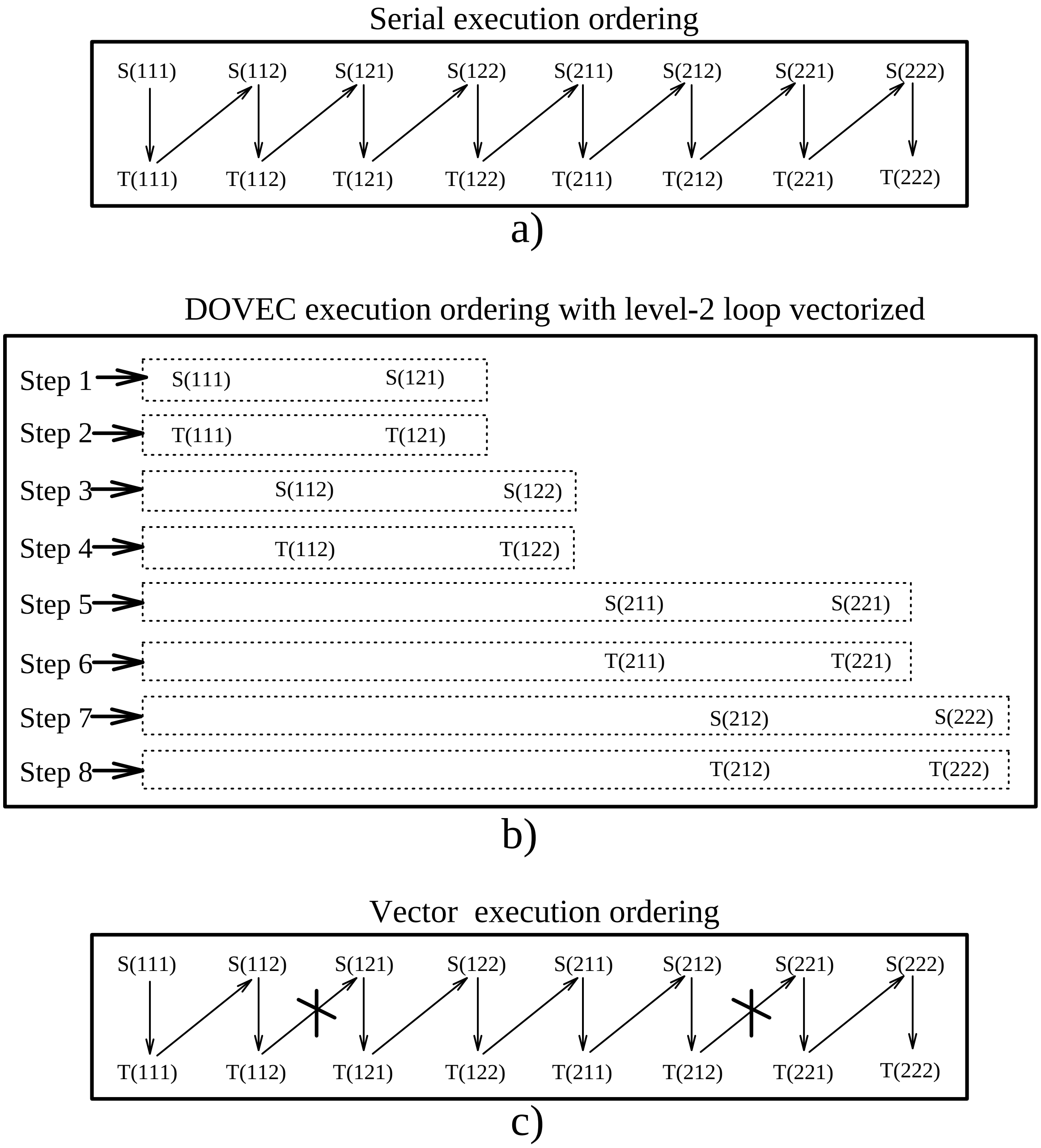

Figure 4.5: Statement ordering for 3 level nest

Proof: Choose an arbitrary $e \in E' - E_k$. Let $e = (S, T, \theta)$. Then choose any $I, J \in \Gamma$ such that $\text{sign}(J - I) = \theta$. We will show that $((S, I), (T, J)) \in \nu_k$; i.e., $S(I) < T(J)$ in the vector execution order ν_k. Let $h = lev(\theta)$. Since $e \notin E_k$, $h < k$ or $h > k$ (and $h \neq \infty$). But by rule 1 of the definition of ν_k, $h \notin \{\infty, k\} \Rightarrow S(I) < T(J)$. □

In essence Theorem 4.1 argues that if a data dependence edge, e, is carried by an outer loop, $L_{lev(e)}$ where ($lev(e) <$ k), the dependence will be satisfied by serial execution of that outer loop. Otherwise, if e is carried by an inner loop ($lev(e) > k$), ν_k will not disturb the serial execution order of its inner loops. In either cases, e can be ignored. For example, consider the nest in Figure 4.6. The nest has a recurrence generated by the inner do-j loop. However $E_1 = \phi$, the recurrence/edge can be ignored.

```
DO I = 1, 100
    DO J = 2, 100
        A(I, J) = f(A(I, J - 1))
    ENDDO
ENDDO
```

Figure 4.6: Vector legality with loop-carried at deeper depth

Given that we know some of the data edges to include for legality test of ν_k, let us now characterize some of the data dependence constraints that will be violated by ν_k.

Definition 4.2 A dependence $S_1\ \delta^*\ S_2$ is *k-vectorization preventing* iff ν_k violates the data dependence.

Theorem 4.2 Let θ be the direction vector associated with a dependence $S\ \delta^*\ T$ in loop L_k. Let i_h, $i \in \theta$, denote repetition of i h times. If h is zero then $i \notin \theta$. θ is *k-vectorization preventing* if θ = (0^{k-1}, 1, 0^h, -1, $\cdots$) where $h \geq 0$.

Proof: Let $I = (i_1, \ldots, i_k, \ldots, i_h, \ldots)$ and $J = (j_1, \ldots, j_k, \ldots, j_h, \ldots)$ be two arbitrary iteration vectors in Γ such that $\text{sign}(J - I) = \theta$. Thus $i_k < j_k$ and $i_h > j_h$. Hence $I \prec_k J$ and $S(I) < T(J)$. We will show that ν_k will execute $T(J)$ before $S(I)$. Let I$' = (i_1, \ldots, i_{k-1}, 0, i_{k+1}, \ldots)$ and J$' = (j_1, \ldots, j_{k-1}, 0, j_{k+1}, \ldots)$. Since $i_l = j_l$ where $1 \leq l < h$ and $j_h > i_h$, we have $J' \prec_h I'$ and by definition 3.3 of ν_k $T(J)$ will be executed before $S(I)$. □

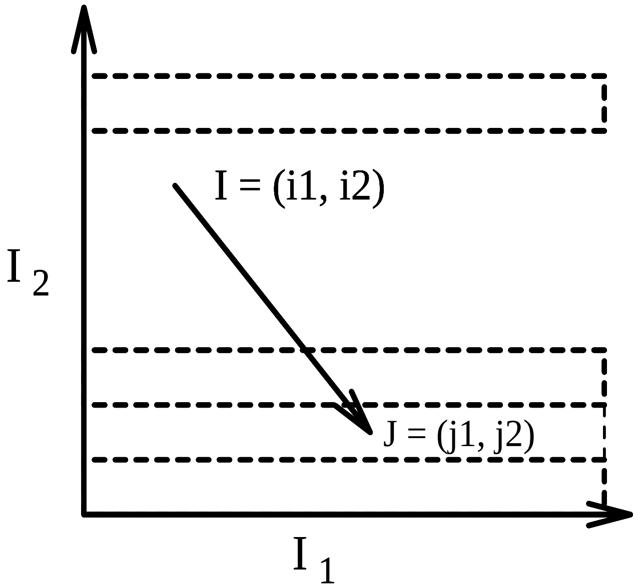

Figure 4.7: Vector Sensitivity Level illustration.

Theorem 4.2 identifies the set of constraints in the iteration space which will be violated by ν_k. These constraints effectively inhibits vectorization along the k-th dimension. Suppose a data dependence constraint imposes an ordering in the iteration space as shown in Figure 4.7, ν_1 will violate the constraint $I \prec_1 J$(with $i_1 < j_1$ and $i_2 > j_2$) by executing J before I.

Lemma 4.1 E_k contains all the potential *k-vectorization preventing* dependencies for ν_k.

Proof: For any arbitrary edge $e = (S, T, \theta)$, we have $lev(\theta) = k$. By definition of E_k, $e \in E_k$. $\square$

Let E_k^p denote a set of *k-vectorization preventing* dependence edge in E_k, $E_k^p = \{\ e \in E_k : e = (S, T, \theta)$ and θ is *k-vectorization preventing* $\}$. The following theorem establishes that a loop L_k is vectorizable if G_{p_k} contains no cycle and and there exists no *k-vectorization preventing* dependence in G_{p_k}.

Theorem 4.3 If there exists no cycle in G_{p_k} and $E_k^p = \phi$ then ν_k will not violate the program semantic.

Proof: Since control dependencies can be converted to data dependencies, without loss of generality we will assume that $E_{p_k} = E_k$. Suppose $E_k^p = \phi$ and ν_k violates the program semantic; we will show that there exists a cycle in G_{p_k}. ν_k violates program semantic imply that there is a serial ordering $S(I) \ll_\theta T(J)$ for some $I, J \in \Gamma$ and $S, T \in V_k$ and this order must be retained because of the presence of dependencies. ν_k, however, does not retain this order, i.e. $((S,I),(T,J)) \notin \nu_k$. By theorem 4.1, $e = (S,T,\theta) \in E_k$. If $S = T$ then (S,S,θ) is a cycle in G_{p_k} and the proof is done. Suppose that $S \neq T$. $lev(\theta) \neq \infty$ since $S < T$ implies $((S,I),(T,J)) \in \nu_k$ (rule 2 of ν_k), contradicting the hypothesis; and $T < S$ is inherently impossible. Hence, $lev(\theta)$ must be k. Given that $e \notin E_k^p$, suppose

case $S < T$: this is not possible since $((S,I),(T,J)) \in \nu_k$ by rule 3.1 and 3.2 of ν_k(and $e \notin E_k^p$).

case $T < S$: for some π in G_{p_k} where $e \in \pi$. If $\pi \notin C(S, G_{p_k})$ then T and S can be reordered since $lev(\theta) = k$; this contradicts the hypothesis. Therefore π must be in $C(S, G_{p_k})$. We conclude that there exists a cycle in G_{p_k}. $\Box$

In Figure 4.8, the outer loop DO-i is not vectorizable because there is a flow backward induced by B. The loop, however, is vectorizable with statement reordering.

```
DO I = 1,100
     DO J = 2,100
        A(I,J) = f(B(I,J))
        B(I+1,J) = g(A(I,J-1))
     ENDDO
ENDDO
```

Figure 4.8: Example code 2 for vector legality illustration.

4.3.4 Vector Selection

The previous section developed a framework for legality of outer loop vectorization. Since there can be only one loop in the nest that can be vectorized, we now focus on the question "which loop to vectorize when there is more than one loop choice?". Vectorizing a loop implies that a vector sequence of vector instructions is generated for that loop. Different loops require different vector sequences and the performance of the loop depends on the selected vector sequence. There are many parameters in each loop contributing to the differences among the loops. We call these parameters *vector variants* and they include:

DO i = 1, 512	tmp(0) = s
s = s + a(i)	DO i = 1, 512, 64
ENDDO	tmp(i:i+63) = tmp(i:i+63) + a(i:i+63)
	ENDDO
	s = tree_height_reduction(tmp(1:64))

Table 4.1: Vectorization of reduction

- Trip count(T): T is the length of the loop.
- Gather/Scatter(GS): GS is indirect load or store, e.g., in A(INDX(I)).
- Conditional(C): C indicates whether or not the vector sequence is conditional.
- Stride-1(V): V is the physical stride between consecutive memory locations.
- Reduction(R): R is a vector code sequence which re-associate the computation, refer Table 4.1 for vector translation schematic of a vector reduction loop.

Let A = { T, GS, C, V, R }. With two loops to choose from, there are $|A| \times |A|$ combinations that need to be considered in order to select which loop to vectorize. We will show a few combinations using small code fragments.

```
REAL a(1000,1000), b(1000,1000), c(1000)
DO i = 1, 10
    DO j = 1, 10
        a(i,j) = a(i,j) + 1
    ENDDO
ENDDO
```

Assume the array element's layout is column-major. Vectorizing the i loop is a better choice because "i" has V (stride one access).

The trip count of the loop, T, is also another important parameter. Suppose that T_j is much larger than T_i,

```
DO i = 1, 10
   DO j = 1, 100
      a(i,j) = a(i,j) + 1
   ENDDO
ENDDO
```

then the cost trade off between V and T must be determined.

Cost trade-off must also be made between a vector loop, i, and a reduction loop, j. For example,

```
DO i = 1, 100
   DO j = 1, 100
      c(i) = c(i) + a(i,j)
   ENDDO
ENDDO
```

vectorizing loop-i is more profitable. However, if T_j is much larger than T_i, for example

```
DO i = 1, 10
   DO j = 1, 1000
      c(i) = c(i) + a(i,j)
   ENDDO
ENDDO
```

then vectorizing loop-j would be more profitable.

Gather/scatter and condition could also have different impact on loop choice, for example

```
DO i = 1, 10
   DO j = 1, 1000
      IF ( c(i) ) THEN
         a(i,j) = b(i,indx(j))
      ENDIF
   ENDDO
ENDDO
```

Overhead	Value	When	Comment
VLO	0	trip $<$ VL	no loop structure
	c1	trip $\geq$ VL	remaining vector and branch back
VSC	c2	always	vector startup cost
ML	VSC	unit stride	memory load
	VSC+c3+c4	reduction	c3: setup, c4:final reduction
	VSC+c5	non-unit stride	Unit stride is used as base
MS	VSC	non-unit stride	memory store

Table 4.2: Source of overhead: proposed handrules

vectorizing loop-i means that the vector accesses are stride-1 but are conditional. While vectorizing loop-j avoids the condition problem, c(i) is invariant with respect to loop-j, but the access to b(i,indx(j)) requires an additional vector load.

4.3.5 Metric

In selecting a loop to be vectorized from a set of n loops, there are $|A|^n$ combinations of cost trade-off that must be considered. We propose a metric which is computationally less intensive than the exhaustive computations of all combinations. This metric computes the overhead cost associated with each loop if it were to be executed as vector. The overhead of each loop is then compared to decide which loop should be vectorized. There are three different sources of vector overhead: vector loop overhead (VLO), vector start up(VSC) for each vector operation, and memory access overhead(MO). VLO is the cost needed to strip-mine the loop when its trip count is larger than the vector length(VL) of the vector register. VSC is overhead needed to initiate the vector instruction. MO is overhead for non-unit stride access.

The *vector variants* defined in A above can be equated to overhead. T and V have direct relation with VLO and MO, respectively. Others need to be converted. C implies the memory operations it controls are being converted to GS. GS is treated as two MO's. R operation translates to an MO plus initial set up cost plus final reduction cost. Table 4.2 shows the summary of the various costs. Unit stride is used as a base unit for memory operation.

We now define the metric as

$$O_l = \frac{VLO + (\lceil(\frac{T_l}{VL})\rceil * (ML + MS))}{T_l}$$

where $\lceil(\frac{T_l}{VL})\rceil$ is the number of vector trips. The metric computes the overhead associated with each loop of interest. Since the metric is measured in term of overhead, preference will be given to loop with lower overhead.

Let us apply the metric to previous examples. The loops are shown on the left and their corresponding overheads, O_l, are shown on right along with selection comments. Assuming that VLO = 20 and VSC = 5.

DO i = 1, 10	$O_i = (0 + 1 * (VSC + VSC))/10$
DO j = 1, 10	$O_j = (0 + 1 * (VSC + c5 + VSC))/10$
a(i,j) = a(i,j) + 1	Loop-i is a better choice since $c5 > 0$.
ENDDO	
ENDDO	

DO i = 1, 10	$O_i = (0 + 1 * (VSC + VSC))/10 = 1$
DO j = 1, 100	$O_j = (VLO + 2 * (VSC + c5 + VSC))/100$
a(i,j) = a(i,j) + 1	$O_j = (40 + 2 * c5)/100$
ENDDO	Loop-j is better if non-unit stride's
ENDDO	latency is less than 30.

DO i = 1, 100	$O_i = (VLO + 2 * (2VSC + VSC))/100 = 50/100$
DO j = 1, 100	$O_j = (VLO + 2 * (VSC + c3 + c4 + VSC + c5 + VSC))/100$
c(i) = c(i) + a(i,j)	$O_j = (50 + c3 + c4 + c5)/100$
ENDDO	Loop-i is better.
ENDDO	

DO i = 1, 10	$O_i = (0 + 1 * (2VSC + VSC))/10 = 15/10$
DO j = 1, 1000	$O_j = (20 + 16 * (VSC + c3 + c4 + VSC + c5 + VSC))/1000$
c(i) = c(i) + a(i,j)	$O_j = (260 + 16 * (c3 + c4 + c5))/1000$
ENDDO	Loop-j is if the overhead of reduction and
ENDDO	non-unit stride for a(i,j) is less than 77.5.

```
DO i = 1, 10
  DO j = 1, 1000
    IF ( c(i) ) THEN
      a(i,j) = b(i,indx(j))
    ENDIF
  ENDDO
ENDDO
```

$O_i = (0 + 1 * (VSC + VSC + c5))/10 = (50 + c5)/10$

$O_j = (VLO + 16(VSC + VSC + c5))/1000$

$O_j = (500 + 16 * c5)/1000$

Loop-j is a better choice.

4.4 Experiments and Observations

This sections presents experimental results of outer loop vectorization opportunity and performance using Perfect Benchmark [9]. We demonstrate the applicability of OLV framework by answering the following questions:

- How often does outer loop vector candidate(OLC) occur
- What is the nature of the loop nest contains an OLC
- How often does vector selection occur
- Performance impact of OLV

4.4.1 Experiment Description

Our environment is a Cray Y-MP8 (shared-memory multiprocessor [43] with eight PEs). Each PE has 64-element vector capability. Total system memory size is 256 MB, arranged in 256 banks. The CPU clock cycle is 6.0 nanosecond. The Unicos operating system runs on this eight-processor platform, which supports Autotasking ^{TM} [42]. CF90^{TM}, the Cray Fortran-90 environment[44], includes state-of-the-art dependency analysis, loop restructuring techniques and can detect many DOALL loops thereby generating parallel SPMD [18] programs. The OLV framework is implemented in the restructuring phase of CF90.

The restructuring phase of CF90 consists many transformations. Among the transformations, two relevant transformations which impact the data being collect for OLV are perfect-nest-creation and loop-collapse[21]. Perfect-nest-creation uses loop distribution to create perfect nest out of non-tightly nested loops. A non-tightly nested loops will be made perfect nested loop if distribution does not require any scalar expansion[50]. Collapse reduces the nesting depth of a loop nest by making the trip count of some loop, L', longer. This transformation always improves the performance of

	Nesting Depth					
	1	2	3	4	5	All
inner candidate	671	369	95	25	4	1164
outer candidate	-	229	72	4	0	305
both candidate	-	225	72	4	0	301
vectorized inner	671	318	88	21	4	1102
vectorized outer	-	55	7	4	0	66
Total vectorized	671	373	95	25	4	1168

Table 4.3: Vector Candidate and Translation by Depth with Collapse

the nest since the resulting collapse nest has less loop control overhead. Furthermore, if longer trip count is desirable if L' is vectorized. Perfect-nest-creation impact the frequency of the number of imperfect nest and loop-collapse reduces the frequency of OLC.

4.4.2 Static Characteristic

We report the static(compile time) characteristics of loop nests without call statement(s). The characteristics include

- Nesting Depth.
- Tight nest(after perfect-nest-creation)
- Number of inner-most candidates(ILC)(at most 1).
- Number of outer candidates(OLC)
- Selection choice: innermost vectorized(ILV) or outer vectorized(OLV)

Table 4.3 shows the vector candidate and vector selection. In 301 of 1168 nests there was a choice between inner and outer loop vectorization [1]. Our system elected to perform outer loop vectorization for 62 of the 301 nests for which both inner and outer loop vectorization was possible. In 4 nests only OLC was available, i.e. the innermost loops were not vectorizable. Figure 4.9 shows a sample of such loop. The inner loop, DO 10, is not vectorizable because there is a recurrence involving F1.

[1] There is an additional 74 OLC opportunities when loop-collapse is disabled.

```
      NX1=NX-1
      NY1=NY-1
      DO 20 J=1,NY
          F1=F(1,J)
          DO 10 I=2,NX1
              F0=0.5*F(I,J)+0.25*(F1+F(I+1,J))
              F1=F(I,J)
              F(I,J)=F0
10        CONTINUE
20    CONTINUE
```

Figure 4.9: Outer loop vector candidate only

OLV Candidates per Nest	0	1	2	3	Total
Number of nests	155	111	24	2	292
OLV (generated)	-	30	4	2	36

Table 4.4: Candidates and Vectorizable of Imperfect Nests

Table 4.4 shows the frequency of OLV contained in non-tightly nested loops. Of the 1168 nests which had at least a vector candidate, there was a total of 292 non-tightly nested loops. Our system elected to perform outer loop vectorization for 36 of the 292 non-tightly nested loops. Also, Perfect-nest-creation created 114 tightly-nested loops.

Table 4.5 shows the distribution of vector selection, i.e., how often does the system has to make a choice. The majority of the nests is single level. The average vectorization choices per nest is 1.3.

Table 4.6 shows vectorization candidate and selection by application. 21.6% of OLC were selected to vectorize, while 94.7% of ILC were selected to vectorize. The high percentage of ILC selection is largely due to inner loop bias of the selection algorithm. Since many applications are tuned for vectorization of innermost loops, i.e., loop nests are written so that the innermost loop will be vectorized by the compiler, the selection algorithm is biased toward selecting the innermost vector candidate when symbolic trip count is encountered.

vector choices	1	2	3	4
nests	867	256	43	2
percentage	74.2	21.9	3.7	0.2

Table 4.5: Distribution of Vector Selection

Benchmark	OLV	OLC	ILV	ILC	OLV/OLC	ILV/ILC
adm	2	38	143	143	5.3%	100.0%
arc2d	1	48	136	137	2.1%	99.3%
bdna	8	8	115	123	0.0%	93.5%
dyfesm	16	23	80	96	69.6%	83.3%
flo52	3	65	100	103	4.6%	97.1%
mdg	0	5	27	27	0.0%	100.0%
mg3d	0	15	54	54	0.0%	100.0%
oceanp	2	22	64	64	9.1%	100.0%
qcd	0	3	86	86	0.0%	100.0%
spec77	29	68	211	240	42.6%	87.9%
spicep	0	0	6	6	0.0%	100.0%
track	5	9	48	53	55.6%	90.6%
trfd	0	1	32	32	0.0%	100.0%
TOTAL	66	305	1102	1164	21.6%	94.7%

Table 4.6: Vectorization Candidates and Selection by Application

```
      SUBROUTINE EX1 (A, B, C)
      DIMENSION A( 12 , 12 ),B( 12 ),C( 12 )
      COMPLEX B, C, SUM
      DO 2 I=1, 12
      SUM = CMPLX(0.,0.)
      DO 1 J=1, 12
      SUM=SUM+CMPLX(A(I,J)*REAL(B(J)),A(I,J)*AIMAG(B(J)))
1     CONTINUE
      C(I) = SUM
2     CONTINUE
      RETURN
      END
```

Figure 4.10: Example Case: Perfect Club Code 1

4.4.3 Speedup

For comparison purposes, we instrumented two compilers CF90-I and CF90-O. CF90-I vectorizes only innermost loop, while CF90-O has outer loop vectorization framework. Performance of the generated code by the two compilers is then compared using different input samples. The input samples include selected subroutines from the Perfect Club. Speedup is computed as $\frac{\text{time of code generated by CF90}-\text{I}}{\text{time of code generated by CF90}-\text{O}}$.

To demonstrate the applicability of OLV, we selected 3 subroutine samples with the following properties:

- The nature of the computation of each routine is unique
- Each routine has at least one outer loop that was vectorized
- The nest involving OLV was non-tightly nested

The speedup for Figure 4.10 is 2.04. CF90-O vectorized the DO-2 loop, while CF90-I vectorized the DO-1 loop. Both loops have the same trip count. Vectorizing the outer loop is better since the inner loop has to perform vector reduction while the outer loop does not.

The speedup for Figure 4.11 is 1.2. CF90-O vectorized the DO-3 loop, while CF90-I vectorized the DO-1 and DO-2 loops. Speedup can mainly be attributed to longer vector length with no additional vector overhead. Note that vectorizing the outer DO-3 is numerically more stable than vectorizing the inner loops since it does not re-associate the computations of CRP, CIP, CRM and CIM variables.

The speedup for Figure 4.12 is 15.24. CF90-O vectorized the DO-4 and DO-8 loops, while CF90-I vectorized the DO-3, DO-7 and DO-6 loops. Speedup can be attributed to much longer vector length, the variable SUM is treated as a vector invariant which results in many vector load and store savings.

4.5 Summary

In this chapter we advocate the idea of vectorizing a non-innermost loop without the application of loop distribution and interchange transformations, i.e., vectorization on non-innermost loops directly . We develop a framework of OLV. First, from the target loop nest we identify the vector execution order that must be followed by the OLV technique. Second, we propose a set of conditions identifying which loop (among the set of loops in the loop nest) is vectorizable in-place and which loop is not vectorizable in-place. If none or a single loop is vectorizable then the choice is obvious, however, if more than one loop is vectorizable then we propose a vector selection criterion using which the most beneficial (per parallelism) loop can be identified for vectorization. Our framework is implemented in $CF90^{TM}$, the Cray Fortran 90 compiler. Experiments were performed on a Cray YMP-8 using a modified version of $CF90$ to allow compilation with or without OLV. OLV opportunity was found in 305 of 1168 vectorizable nests in the Perfect Benchmark, with OLV actually selected for 66 nests. Significant(up to 15×) speedup at the subroutine level of some routines were also observed.

```
      SUBROUTINE EX2(FLN,AP,AM,PLN)
C     SR COMPUTES FOURIER MULTIPLIERS
C     IN AP COEFFS. OF SYMMETRIC LEGENDRES
C     IN AM COEFFS. OF ANTISYM. LEGENDRES
      DIMENSION AP( 31 ),AM( 31 ),
      1PLN( 32 , 31 ),FLN( 31 , 31 )
      COMPLEX AP,AM,FLN

      DO 3 L=1, 31
      CRP=0.
      CIP=0.
      DO 1 I=1, 31 ,2
      CRP=CRP+PLN(I,L)* REAL(FLN(I,L))
      CIP=CIP+PLN(I,L)*AIMAG(FLN(I,L))
1     CONTINUE
      CRM=0.
      CIM=0.
      DO 2 I=2, 30 ,2
      CRM=CRM+PLN(I,L)* REAL(FLN(I,L))
      CIM=CIM+PLN(I,L)*AIMAG(FLN(I,L))
2     CONTINUE
      AP(L)=CMPLX(CRP+CRM,CIP+CIM)
      AM(L)=CMPLX(CRP-CRM,CIP-CIM)
3     CONTINUE
      RETURN
      END
```

Figure 4.11: Example Case: Perfect Club Code 2

```
      SUBROUTINE EX3(F,EIGG,EIGGT,DOTPRO,MODS,INOUT)
      DIMENSION EIGG( 12 , 12 ),EIGGT( 12 , 12 ),DOTPRO( 12 )
      COMPLEX F( 961 , 12 ),COL( 12 ),SUM
C     INOUT=-1 TO OBTAIN VERTICAL MODE EXPANSION
C     INOUT=+1 TO OBTAIN SPET. COEFS. FROM VERTICAL EXPANSION
      IF(INOUT)10,20,30
10    CONTINUE
      DO 4 I=1, 961
      DO 2 MOD=1,MODS
      SUM=(0.,0.)
      DO 1 LEV=1, 12
      SUM=SUM+CMPLX(EIGGT(LEV,MOD)* REAL(F(I,LEV)),
      1        EIGGT(LEV,MOD)*AIMAG(F(I,LEV)))
1     CONTINUE
2     COL(MOD)=CMPLX(DOTPRO(MOD)*REAL(SUM),DOTPRO(MOD)*AIMAG(SUM))
      DO 3 MOD=1,MODS
3     F(I,MOD)=COL(MOD)
4     CONTINUE
20    RETURN
30    CONTINUE
      DO 8 I=1, 961
      DO 5 MOD=1,MODS
5     COL(MOD)=F(I,MOD)
      DO 7 LEV=1, 12
      F(I,LEV)=(0.,0.)
      DO 6 MOD=1,MODS
      F(I,LEV)=F(I,LEV)+CMPLX(
      1        EIGG(LEV,MOD)*REAL(COL(MOD)),EIGG(LEV,MOD)*AIMAG(COL(MOD)))
6     CONTINUE
7     CONTINUE
8     CONTINUE
      RETURN
      END
```

Figure 4.12: Example Case: Perfect Club Code 3

Chapter 5

Implementation

5.1 Introduction

In this chapter we discuss the implementation details necessary to put the results developed in earlier chapters into product. The actual implementation resulted in thousands of lines of code written in C. In the process many interesting engineering solutions were developed. We report two significant engineering solutions developed to carry out the implementation in a product compiling system.

The first problem, conditional privatization, arises when the write/read access is conditionally defined or referenced. Moreover the semantic of a Fortran do loop requires generation of an IF statement to check if the loop is to be executed at all. This additionally generated IF check generated for each do loop forces the private array analysis phase to perform containment analysis in the presence of IF statements, even when the original program doesn't contain any IF statement.

The second problem is to perform vector code translation for in-place outer vectorized loops. As will be shown later, vector translation technique for innermost vectorized loops is inadequate. More specifically, an in-place outer vectorized loop has one or more serial loops nested inside. These serial loops have local computations that must not be mapped to vector forms. Thus the vector code generator must recognize these computations to avoid translating them into vector instructions.

The remainder of this chapter is divided into 4 main sections. Section 5.2 describes the intermediate representation of a do-loop generated by the respective C or FORTRAN front-end, and loop normalization format. Section 5.3 describes the conditional privatization problem in detail and proposes solutions. Section 5.4 gives an overview of a vector code generator to generate vector code for the innermost loop and show how it can be extended to generate vector for outer loops. The chapter closes with a summary section.

5.2 Loop Representation

Given a Fortran do loop of the form

DO $I = start, end, incr$

$\ldots$

$\ldots A(I) \ldots$

$\ldots$

ENDDO

we refer to I as the induction variable of the loop. The semantic of the loop, stated by Fortran standard, does not require the loop body be executed. In other words, the trip count of the loop may be zero and execution of the loop body will be skipped. This means that a compiler must generate a precondition, a check before each loop, to ensure that the loop will not be executed if there is no work. In our system, this check is being generated by the front-end and the intermediate code for the above loop is as follows:

$$
\begin{array}{ll}
& \text{trips} = \text{abs}((end - start - incr)/incr) \\
& \text{IF (trips} > 0 \text{) THEN} \\
& \quad I = start \\
\text{loop:} & \\
& \quad \ldots \\
& \quad \ldots A(I) \ldots \\
& \quad \ldots \\
& \quad I = I + incr \\
& \quad \text{IF (} I \leq end \text{) GOTO loop} \\
& \text{ENDIF}
\end{array}
$$

We say that a loop is normalized [50] [51] when the induction variable starts at zero and the increment is one. By normalizing the loop, the "*incr*" expression specified on the do statement is pushed into each individual subscript. For example, the previous loop is normalized as

$$
\begin{array}{ll}
& \text{trips} = \text{abs}((end - start - incr)/incr) \\
& \text{IF (trips} > 0 \text{) THEN} \\
& \quad I = 0 \\
\text{loop:} & \\
& \quad \ldots \\
& \quad \ldots A(incr * I + start) \ldots \\
& \quad \ldots \\
& \quad I = I + 1 \\
& \quad \text{IF (} I \leq \text{trips) GOTO loop} \\
& \text{ENDIF}
\end{array}
$$

5.3 Conditional Privatization

The containment question $R \subseteq W$ is much more difficult to determine in the presence of IF statements with condition value that is not constant at compile time. Difficulty arises when the W and/or R sets are conditionally defined, which means that the set may or may not be written and/or read at run time.

In the containment question, the goal is to produce the largest possible W and the smallest possible R so as to maximize privatization. It is, however, acceptable to overestimate R and underestimate W. When a read site is conditionally executed, we can overestimate by ignoring the condition and test for containment just as if it is unconditionally executed. Similarly, a write set can be underestimate by ignoring the write sites which are conditionally executed. A conservative compiler approach is to consider only the unconditionally written sets by ignoring the conditionally defined ones. Unfortunately most write sites are conditionally defined since Fortran do loop semantic requires generating an IF statement at the beginning of each loop as shown in section 5.2, this approach greatly underestimates the W sets, and as a result the possibility of finding $R \subseteq W$ is significantly reduced. This stands as a potential reduction of array privatization, even though the array is privatizable. Note that a loop with constant start, end and increment will result in a constant trip count. Based on the value of the trip count, the compiler can decide whether or not to generate the loop body. It is therefore unnecessary to generate the precondition check.

5.3.1 Conditional Percolation

The problem of computing the write set for an IF statement is similar to the available-expression problem in classical data flow. The confluence operator is intersection to ensure that the write set is available regardless of which branch taken. In particular, the write set produces by an IF-THEN-ELSE construct is the intersection of the write set in the THEN body and the write set in the ELSE body. The write set in either branch can be the union of many write sets.

For a nested IF structure, the write sets of the innermost IF statement are formed by taking the union of the individual write set in each respective THEN body and ELSE body. The resulting sets are then intersected, and the result, r, is then used to compute the write set for next outer IF level. At that level, r is part of b where b is either the THEN body or ELSE body of the next outer IF level. We then take the union of r with other write sets in b and the result is then intersected with the union of the write sets on the other branch to move outward to the next IF level. The process repeats until the outermost IF statement is reached, at which point, a single write set is produced.

If the presence of conditional, the write set $\bar{W_i}$ of equation(2) in chapter 3 has to be redefined. $\bar{W_i}$ now consists of unconditional and conditional write sites. Function *private* shown in Figure 5.1 performs containment testing for $R_i \subseteq \bar{W}_i$, it will take the union of the unconditional write sites and intersect the conditional ones. The loop at line 12 computes $\bar{W}_i$, invoking *percolate* to compute the write set for both conditionally and unconditionally defined ones. *percolate* invokes itself recursively, synthesizing the write set upward. The if check at line 1 returns an empty write set for missing branch. Once the $\bar{W_i}$ set is computed, containment test is performed at line 17.

Figure 5.2 shows a code sample to illustrate the working details of *private* function. The corresponding PDG structure of the program is shown in Figure 5.3. The result produces by each node

```
function percolate( n : Node ) return Write_set is
    (1) if n ≠ null then
    (2)     case n.kind of
    (3)         IF:
    (4)             return percolate(n.true) ∩ percolate(n.false)
    (5)         SEQ: /* Then body or Else body */
    (6)             return ⋃_{m∈n.children} percolate(m)
    (7)         ASSIGN:
    (8)             return n.write_set
    (9)     end case
    (10) return ∅
end percolate
function private(r, L : Node ) return Boolean is
    /* the array being privatized is implicit here. */
    /* r contains read site to test for privatization */
    /* L candidate loop to make r private of */
    (11) W := ∅
    (12) for m ∈ L.children loop
    (13)        if lexically_preceding(m, r) then
    (14)            W := W ∪ percolate(m)
    (15)        end if
    (16) end loop
    (17) if R_r ⊆ W) then
    (18)        return true
    (19) return false
end private
```

Figure 5.1: Percolation Algorithm

```
DO i = 1, n
    IF ( c1 ) THEN
        W1
        IF ( c2 ) THEN
            W2
        ENDIF
    ELSE
        W3
    ENDIF
    W4
    IF ( c3 ) THEN
        R
    ENDIF
ENDDO
```

Figure 5.2: Percolation example

is displayed on the right hand side of that node. The first call to *percolate* produces $W_1 \cap W_3$. The second call produces W_4. The final value of W is $(W_1 \cap W_3) \cup W_4$.

5.3.1.1 Limitation

Consider an IF instruction like

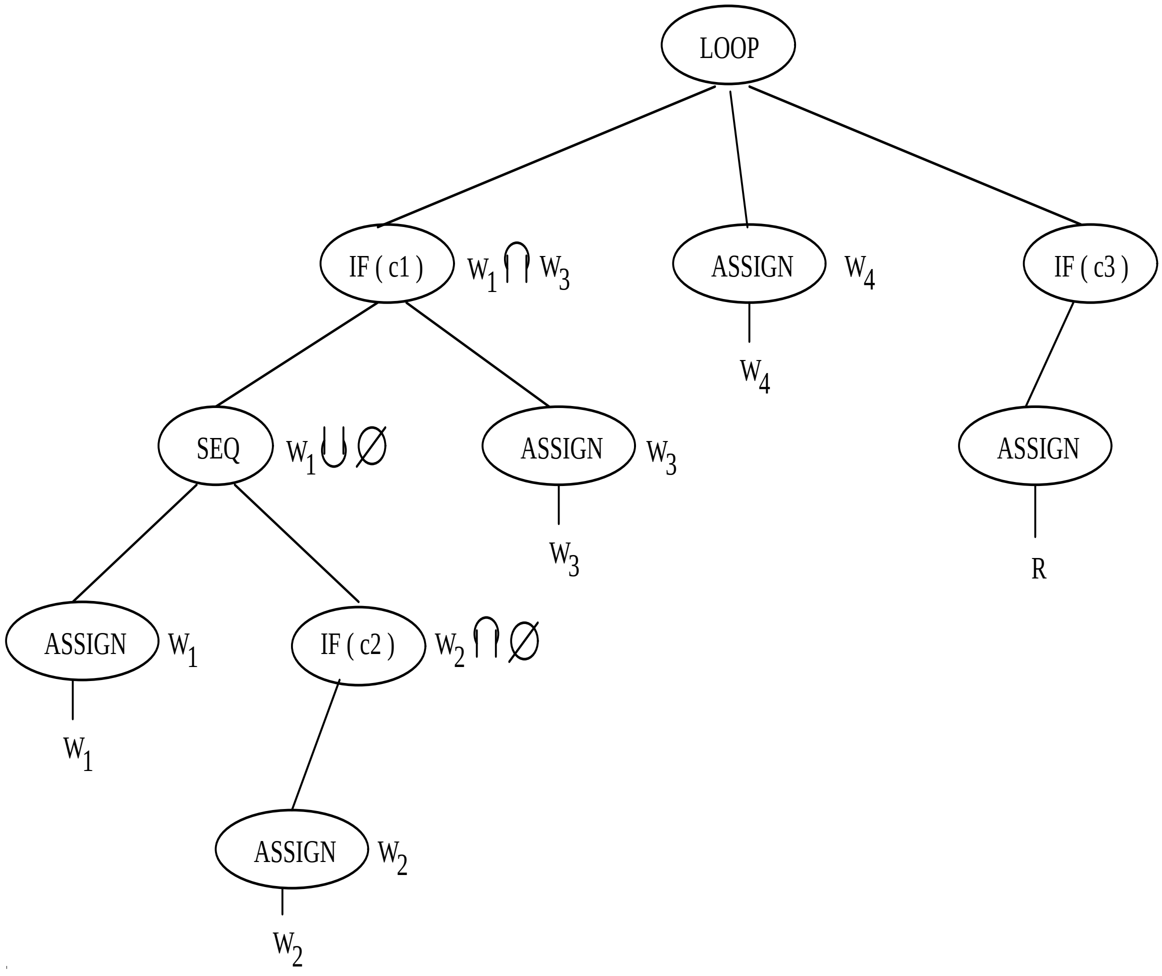

Figure 5.3: Example of Percolation

```
IF (condition) THEN
    W1
    ...
ELSE
    W2
    ...
ENDIF
```

Since the outcome of the condition in the IF statement is not known at compile time and percolation does not attempt to determine whether the then-path or the else-path will be executed, percolation estimates the W set as $(W_1 \cap W_2)$. Note that we trivially have $(W_1 \cap W_2) \subseteq W_1$ and $(W_1 \cap W_2) \subseteq W_2$. In general, an IF instruction without an ELSE part will result in $W_2 = \emptyset$. Similarly an IF instruction without a THEN part will result in $W_1 = \emptyset$. Thus in a one branch IF instruction(without ELSE or without THEN part), the set "$W_1 \cap W_2$" is empty.

Percolation algorithm can sometimes underestimate the W set and thus reducing the possibility of finding the potential private array. Consider the following example

```
DO k = 1, m
    DO i = 1, n
        A(i) = ...
    ENDDO

    DO i = 1, n-1
        ... = A(i)
    ENDDO
ENDDO
```

suppose that we want to privatize A for the k-loop. Based on the loop structures, the write set of A in the first loop contains the read set of A in the second loop, $R \subseteq W$. However, the intermediate representation shown in the following shows that A is conditionally executed by the IF(n-1 > 0) statement. This IF statement does not have an ELSE clause and thus *percolate* produces an empty write set for A, i.e. $W = \emptyset$ since $W_2 = \emptyset$. This in turn will make the containment test to fail, i.e. $R \not\subseteq \emptyset$.

```
IF ( m-1 > 0 ) THEN
```

```
        k = 0
loop_k:
        IF ( n-1 > 0 ) THEN
             i = 0
        loop1:
             A(i+1) = ...
             i = i + 1
             IF ( i ≤ n-1 ) GOTO loop1
        ENDIF
        IF ( n-2 > 0 ) THEN
             i = 0
        loop2:
             ... = A(i+1)
             i = i + 1
             IF ( i ≤ n-2 ) GOTO loop2
        ENDIF
        k = k + 1
        IF ( k ≤ m-1 ) GOTO loop_k
    ENDIF
```

5.3.2 Pruning

Our aim in this subsection is to improve the probability of successful privatization by enlarging the estimate-write set. Enlargement is done by employing a pruning method to reduce the number of conditions. We first make several observations regarding the lexical position of the R and W set occurrences. These observations allow us to build improved techniques, e.g., to avoid the naive W set intersection approach - if possible. The idea can be illustrated by considering the following IF statement

```
IF (condition) THEN
     W_1
     R
     ...
ENDIF
```

Here the W set for the *else*-part is null. If we intersect the W sets for *then* and *else* parts, it results in a $\emptyset$ which certainly does not contain R. However, it is trivial to observe that if $R \subseteq W_1$ then the read set is properly contained. Thus, it is not necessary to *first* intersect the W sets and *then* carry out the containment check. Since, the occurrence of R and W_1 are together (i.e., under the same condition), we can *first* afford to check the containment.

The above observation can be extended to more general cases. We now consider two IF statements.

$$\text{IF (condition 1) THEN } W_1 \text{ ENDIF}$$
$$\text{IF (condition 2) THEN } R \quad \text{ENDIF}$$

Suppose, i) the variables used in "condition 1" have not been redefined after executing the first IF statement,i.e., "condition 1" is *available* at the second IF statement *and* ii) "condition 2" $\Longrightarrow$ "condition 1", then whenever R is executed W_1 is also executed. In other words, if $R \subseteq W_1$ then we can safely conclude that *read is contained by the write.* Note that, the naive method would have produced a null estimate W set from the first IF-statement, and failed to contain R in the second IF statement.

The above example illustrates the importance of being able to establish the implication between different condition structures, there we have "condition 2" $\Rightarrow$ "condition 1". The implication does not require the value of "condition 1" and "condition 2" to be known at compile time. Knowledge of the implication enables the compiler to ignore/prune one branch of the IF statement. In this example, the ELSE part of the first IF statement is irrelevant. We summarize the above observations with the following algorithm for pruning: Given an initial condition C_I and an IF statement

$$\text{IF (} c \text{) THEN } W_1 \text{ ELSE } W_2 \text{ ENDIF}$$

Denote the estimate write set for the above IF statement by W.

- if $C_I \Longrightarrow c$ then $W = W_1$
- if $C_I \Longrightarrow \neg c$ then $W = W_2$
- if $C_I \not\Longrightarrow c$ and $C_I \not\Longrightarrow \neg n_c$ then pruning does not occur and we set $W = W_1 \cap W_2$.

It is clear that pruning yields a larger(better) estimate write set for a conditional write site.

This pruning process can be further extended to the case when the IF structure containing the W sets is nested. In this case the conditions "condition 1" and "condition 2" are expressed in conjunctive form. More specifically, in order for a statement S in a nested IF structure to be executed all conditions on the path leading to it must be met. To be more precise, let c_i denote the condition of an IF statement S_c on the path to S. Define

$$\tilde{c}_i = \begin{cases} c_i & \text{if } S \in \text{THEN body of } S_c \\ \neg c_i & \text{if } S \in \text{ ELSE body of } S_c \end{cases}$$

The condition for executing S is of the form $\tilde{c}_1 \wedge \tilde{c}_2 \wedge \ldots \wedge \tilde{c}_s$ where $s \geq 0$. By convention $s = 0$ means that execution of S does not depend on any condition. The conditions for executing R and W in a nested IF structure are then of the form $C_r = \tilde{c}_1 \wedge \tilde{c}_2 \ldots \wedge \tilde{c}_{n_r}$ and $C_w = \tilde{c}'_1 \wedge \tilde{c}'_2 \ldots \wedge \tilde{c}'_{n_w}$ respectively.

To prune a nested IF structure, first we determine the condition C_R for reading R. We set the initial condition $C_I = C_R$ then start pruning from the highest level condition flow and down the condition flow hierarchy. At each condition node n with condition n_c, we use C_I which consists of all information gathered prior to execution of n, to determine whether or not n_c is decidable. Decidability is based on successful evaluation of either $C_I \Longrightarrow n_c$ or $C_I \Longrightarrow \neg n_c$. If n_c is decidable then we can prune one branch of the IF statement. Suppose we reach an IF statement of the form

$$\text{IF } (n_c) \text{ THEN } \ldots \text{ ELSE } \ldots \text{ ENDIF}$$

where $C_I \not\Longrightarrow n_c$ and $C_I \not\Longrightarrow \neg n_c$. Then we say that the condition n_c is *undecidable*. At this stage we are forced to use the approximation

$$W = W_1 \cap W_2$$

where W_1 = write set of THEN body and W_2 = write set of ELSE body Pruning process starts all over again with two distinct branches to estimate W_1 and W_2.

1. The THEN branch will have for initial condition $C_I = C_I \wedge n_c$
2. The ELSE branch will have for initial condition $C_I = C_I \wedge \neg n_c$

The following theorems show that pruning results in a larger estimate write set from percolation.

Theorem 5.1 Let S be a nested IF structure. Let $P_1(S)$ and $P_2(S)$ denote the structures obtained after pruning using initial conditions C_{I_1} and C_{I_2} respectively. Let W_1 and W_2 be the write sets obtained by percolating $P_1(S)$ and $P_2(S)$ respectively. If $C_{I_1} \Longrightarrow C_{I_2}$ then $W_2 \subseteq W_1$.

Proof: Let n be a node with condition n_c. If n_c is decidable with respect to C_{I_2} then either $C_{I_2} \Longrightarrow n_c$ or $C_{I_2} \Longrightarrow \neg n_c$. Since $C_{I_1} \Longrightarrow C_{I_2}$ we must also have $C_{I_1} \Longrightarrow n_c$ or $C_{I_1} \Longrightarrow \neg n_c$. Thus n_c is decidable with respect to C_{I_1}. Now let $\mathcal{F}_1$ be the family of decidable nodes of $P_1(S)$ and $\mathcal{F}_2$ be the family of decidable nodes of $P_2(S)$. From the above observation we have $\mathcal{F}_2 \subseteq \mathcal{F}_1$. Now since a node with a decidable condition is pruned, percolation will result in a larger write set for that node. Since $\mathcal{F}_2 \subseteq \mathcal{F}_1$ we conclude that $W_2 \subseteq W_1$. $\Box$.

The above theorem suggests that we make use of the strongest hypothesis available for pruning. For the purpose of containment analysis, one can use C_r as the starting "initial" condition. Even in the event that the read site is unconditionally defined, i.e., $C_r = \emptyset$, pruning can still occur due to updating of the initial condition. In this case the estimate write set can still potentially be larger than the write set obtained by no pruning. Note that no pruning is equivalent to an empty initial condition and not updating, or all inferences are undecidable. We summarize with the following theorem without proof.

Theorem 5.2 Let S be a nested IF structure. Let $P(S)$ denote the resulting structure after pruning(with any initial condition C_I). Let W_2 be the write set obtained by percolating S. Let W_1 be the write set obtained by percolating $P(S)$. We have $W_2 \subseteq W_1$.

The new conditional privatization algorithm is shown in Figure 5.4. Function Estimate produces a W for each specified PDG substructure n. It recursively descends the substructure headed by n to compute the W set. At each node, local computations are applied to the results returned from its immediate lower level, if necessary, to produce a new result. This new result is then returned to the caller(or parent node). Local action to compute W depends on the node type and are classified as follows:

1. IF : if the condition associated with IF statement is decidable then one branch is produced, i.e., the resulting estimate produced by either the THEN branch or ELSE branch is returned.

   Otherwise the results of the THEN and ELSE branches(returned by recursive calls with each branch as an argument along with a new initial condition corresponding to the branch) are intersected.

2. SEQ: the result is produced by taking the union of all children

3. ASSIGN: a write set is returned. Note that the write set may be empty.

We now present a detailed example as shown in Figure 5.5 to illustrate the pruning algorithm execution.

Percolation: R is compared with $(w_4 \cap (w_1 \cap (w_2 \cap w_3)))$.

Conditional Privatization: Since, $c_3 \implies c_1$, IF at line 1 would flow along the "THEN" part pruning w_5. No information is available between c_3 and c_2, and hence an intersection of the "THEN" function Estimate_W(c : Condition; n : Node) return Write_set is

(1) if $n \neq null$ then

(2) case $n.kind$ of

(3) IF:

(4) if $(c \Rightarrow n.c)$ then

(5) return $Estimate\_W(c, n.true)$

(6) if $(c \Rightarrow \neg n.c)$ then

(7) return $Estimate\_W(c, n.false)$

(8) return $Estimate\_W(c \wedge n.c, n.true) \cap Estimate\_W(c \wedge \neg n.c, n.false)$

(9) SEQ:

(10) return $\bigcup_{m \in n.children} Estimate\_W(c, m)$

(11) ASSIGN:

(12) return $n.write\_set$

(13) end case

(14) return $\emptyset$

end Estimate_W

function $private$(r, L : Node) return Boolean is /* the array being privatized is implicit here. */

/* r contains read site to test for privatization, L candidate loop to make r private of */

(11) $W := \emptyset$

(12) for $m \in L.children$ loop

(13) if $lexically\_preceding(m, r)$ then

(14) $W := W \cup Estimate\_W(C_r, m)$

(15) end if

(16) end loop

(17) if $R_r \subseteq W$ then return $true$

(18) return $false$

end $private$

Figure 5.4: Conditional Privatization Algorithm

1: IF (c_1) THEN

2: IF (c_2) THEN W_1;

3: ELSE If (c_4) THEN W_2;

4: ELSE W_3;

5: ELSE W_4;

6: IF (c_3) THEN R;

Figure 5.5: Pruning algorithm execution, where $c_3 \Longrightarrow c_1$, $\neg c_2 \Longrightarrow c_4$.

and “ELSE” parts of IF at line 2 is activated. Again, since $\neg c_2 \Longrightarrow c_4$, we have only the “THEN” part of IF at line 3 executed (pruning w_3).

The intersection result at line 2 equals $(w_1 \cap w_2)$. This result is percolated to line 1. Hence, R would be compared with $(w_1 \cap w_2)$. Note that $(w_1 \cap w_2)$ is much larger than $(w_4 \cap (w_1 \cap (w_2 \cap w_3)))$, and thus the chance of read being contained by the write is increased.

5.4 Vector Code Generation

5.4.1 Vector Instruction Model

The vector instruction set we use to generate vector code is shown in Figure 5.6. $Vreg \leftarrow Broadcast(scalar)$ creates a vector of scalar elements. A linearized array A(coefficient*i + offset), where i is the vectorizing induction variable, is translated to a vector access tuple (A+offset, coefficient). $< mask >$ is the indices of the vector register whose values are satisfied by the condition specified by $< COND >$; it is used to generate vector code for IF-THEN-ELSE construct. $< IND\_VEC\_REF >$ is used to access memory non-uniformly, e.g., A(INDX(i)). The length of Vreg is set separately by the VL setting instruction.

5.4.2 Innermost Vector Translation Overview

Once an innermost scalar loop is identified as vectorizable, an innermost loop vector(ILV) code generator translates that scalar loop into a vector loop suitable for vector execution. For readability

$VL \leftarrow value$

$Vreg \leftarrow \; <VS> op \;\; Vreg$

$Vreg \leftarrow Broadcast(scalar)$

$VS ::= Vreg|scalar$

$<mask> \; \leftarrow Vreg, <COND>, 0$

$Vreg \leftarrow Memory$

$Memory \leftarrow Vreg$

$Memory ::= <VEC\_REF> | <IND\_VEC\_REF>$

$VEC\_REF ::= (<base>, <stride>)$

$<IND\_VEC\_REF> ::= <base, VEC\_REF>$

$<mask> ::= true \; index$

$<COND> ::= \;\; < | \leq | > | \geq | = | \neq$

$<base> ::= array \; variable$

$<stride> ::= skipped \; distance \; between \; memory \; elements$

$<scalar> ::= scalar \; register$

Figure 5.6: Generic Vector Instruction set

```
DO i = 1, n
    a(i) = b(i) + 1
ENDDO

DOVEC i = 1, n, VL
    a(i: min(i+VL,n)) = b(i: min(i+VL,n)) + 1
ENDVEC
```

Figure 5.7: Vector translation using array syntax notation

of the presentation, we use a dovec construct and array syntax notation to describe the generated vector code. A dovec is a short hand to denote the code sequence to compute the number of vector trips. An array syntax A(start-expression : end-expression) is a short hand for a vector access "A" of length equal to end-expression - start-expression + 1. Figure 5.7 shows the vector translation of a loop using array syntax notation. In that figure, VL is the length of the vector register. b(i: min(i+VL,n)) describes a vector access of b starting from b(i) to b(i+VL) or b(n) which ever is smaller. The min operation is used to ensure the last vector segment access does not over index memory.

A vectorizable loop has many different scalar constructs, each of which requires different vector form. More specifically, a vector code generator has to produce different vector instruction sequences/structures for different scalar construct. To handle the wide variety of scalar constructs, the internal structure of a typical vector code generator incorporates two different phases: *vector classification* and *vector translation*. The first phase, vector classification classifies what vector forms each scalar reference/construct should be translated to. Vector translation, the second phase, uses this information to generate the appropriate vector form for each scalar reference. Vector translation also generates vector looping structure by using strip-mining to partition the iteration space into segments of vector iteration.

There are three different vector classes in the ILV classification. Before describing these classes in detail, we first state the necessary definitions and conventions. Let $\mathcal{L}$(A,IDX) define the nesting level of the index reference IDX with respect to A, where A is an array and IDX is a simple variable. If IDX is not referenced by A then $\mathcal{L}(A,IDX) = 0$. For example, given A(2), A(I), A(B(I)), A(B(C(I))), their nesting level $\mathcal{L}$(A,I) are 0, 1, 2, and 3 respectively. Let vector index(VIX) be the induction variable of the vectorizing loop. The area before the "dovec i = 1, n, VL" is called the *vector-preamble*; similarly the area after "endvec" is called the *vector-postamble*. We now categorize each class and briefly describe their associated code translation.

- **VAR** A vector array reference(VAR) is an array reference subscripted by VIX. Thus by definition, $\mathcal{L}$(A,VIX) is greater than zero. if $\mathcal{L}$(A,VIX) is one, then a VEC_REF is generated. Otherwise a IND_VEC_REF is generated.

- **COND** is an IF statement. If the condition does not contain any VEC_REF or IND_VEC_REF, then scalar code is generated for the condition. Otherwise, a mask is generated for the condition. The true-body execution is subject to the density of the mask. Similarly, the false-body execution depends on the density of the mask's complement. Exclusion is done by compressing VEC_REF or IND_VEC_REF in the IF body to access elements indicated by true or false mask for true-body or false-body respectively. For example, consider the following code segment to be vectorized

```
IF( A(i) .EQ. 0 ) THEN
        ... = B(i)
ELSE
        ... = C(i)
ENDIF
```

  Assume that the vector length is 10 and the vector index begins at 0 and A(i) .EQ. 0 is true for i = 1 and i = 5. Thus the true mask consists of {1,5}, while the false mask consists of {0,2,3,4,6,7,8,9}. The vector length for the true-body is 2 and the VEC_REF for B(i) is converted a non-uniform vector reference, IND_VEC_REF, whose vector index contains 1 and 5. Similarly, the vector length for the false-body is 8 and the VEC_REF for C(i) is converted to an IND_VEC_REF whose vector index contains {0,2,3,4,6,7,8,9}.

- **Non-VAR** A simple reference or an array reference A, where VIX is not referenced in A, so that $\mathcal{L}$(A,VIX) = 0. References in this class are candidates for vector temporary expansion. Vector temporary expansion allows different values computed in different loop iteration to be captured in a vector for later use. Thus it removes anti-dependence and output-dependence dependencies induced by the scalar reference. When a variable is expanded into a vector temporary, copy-out code is needed to capture the last value produced by serial execution. Note that copy-out code is not needed if the reference is not live, or not used, after the loop. Not all Non-VAR references are expanded into vector temporaries, expansion depends on pattern of reference and are classified as follows:

- **DNU** Define but no use: generates copy out code in the vector-postamble.
- **USE** read only(there is no definition of the variable in the loop): generates scalar or generates broadcast code to replicate the scalar value into a vector.
- **DBU** Define before use(within the loop body). There exists a definition of A which dominates all references: expands A into a vector temporary and generates copy-out in the vector-postamble.
- **RED** special recurrence such as reduction: expands into a vector temporary, adds necessary set up in the vector-preamble and cleans up code in the vector-postamble.

5.4.3 Outer Loop Vector Codegen

The traditional way of vectorizing an outer loop is to first use interchange to make that loop the innermost loop, and then vectorizing the resulting innermost. "In-place" outer vector loop vectorization is to vectorize an outer loop without interchanging. An innermost vector loop is a vector loop which operates with scalar operands. On the other hand, the code body of the innermost loop, whose outer parent loop was vectorized, is a serial loop which operates with vector operands. Figure 5.8 shows a code sample of adding a vector to a matrix. Figure 5.9 shows translation code for vectorizing the inner-loop. The "j" loop executes two vector trips with b(i) being referenced as a scalar(since b(i) is invariant with respect to the "j" loop). Figure 5.10 shows the translation code for vectorizing the outer "i" loop. The inner "j" loop executes serially but the operands within its body are vectors.

Most of the vector classes identified in vector classification for innermost-loop-vectorization approach are applicable to OLV approach. The two approaches differ in their vector loop choice; different vector loop choice implies different VIX . By definition of VAR and Non-VAR, different VIX may change a reference to be classified as VAR in one case, but Non-VAR in the other, and vice versa. Thus, for an innermost loop body, different VIX will produce different set of vector and scalar references. For example, consider the reference b(i) in Figure 5.8. If the outer loop is vectorized, VIX is "i" and $\mathcal{L}$(b,i) = 1, b(i) is therefore a VAR. However, if the inner loop is vectorized, VIX is "j" and $\mathcal{L}$(b,j) = 0, b(i) is a Non-VAR. COND classification is also applicable OLV since the translation depends on the occurrence of VAR or Non-VAR in the condition. The detailed classification of Non-VAR, however, has to be extended for OLV approach.

```
DO i = 1, 128
    DO j = 1, 128
        a(i,j) = a(i,j) + b(i)
    ENDDO
ENDDO
```

Figure 5.8: Matrix Vector Addition Example

```
DO i = 1,128
    DOVEC j = 1, 128, 64
        a(i,j:j+64) = a(i,j:j+64) + b(i)
    ENDVEC
ENDDO b) vectorizing inner-loop
```

Figure 5.9: Vectorizing Innermost Loop

```
DOVEC i = 1,128, 64
    DO j = 1, 128
        a(i:i+64,j) = a(i:i+64, j) + b(i:i+64)
    ENDDO
ENDVEC
```

Figure 5.10: Vectorizing Outer Loop

```
DO i = 1,64
    sum = 0
    DO j = 1,64
        a(i,j) = b(j)
        sum = sum + d(i,j)
    ENDDO
    c(i) = sum
ENDDO

        a)
```

```
        i = 1
outer:
        sum = 0
        j = 1
inner:
        a(i,j) = b(j)
        sum = sum + d(i,j)
        j = j + 1
        if (j ≤ 64 ) goto inner
        c(i) = sum
        i = i + 1
        if ( i ≤ 64 ) goto outer
            b)
```

Figure 5.11: Code example 2

In order to generate vector code for outer loops, Non-VAR classification has to be extended. By vectorizing an outer loop of a loop nest in place, the inner loops of that loop execute serially. The inner serial loop allows reference patterns that do not occur with the ILV approach. Consider the code sample in Figure fg:olv-ex2aa) where vector translation is being done for the outer loop, as shown in Figure fg:olv-ex2c. The intermediate generated for the nest is shown in Figure 5.11b). For vector classification, "j" is varied with respect to the "i" loop since j is incremented at the end of each trip of the inner loop. The reference structure of j includes initializing "j" to 1 in the pre-amble of "j" loop, referencing to "j" in the body and redefining(to do the increment). Using ILV definition of Non-VAR ($\mathcal{L}$(j,i) = 0), this structure is DBU, which means that "j" should be expanded into a vector temporary. However, "j" should remain scalar. On the other hand, "sum" has a similar referencing pattern but has been expanded into a vector temporary. Also, b(j) is still scalar ($\mathcal{L}$(b,i) = 0 and read only).

Extension of Non-VAR classification for OLV requires knowledge whether a given statement is involved in a vector operation or not. As demonstrated earlier, this is necessary because a statement could strictly be scalar. A classification algorithm to perform outer vector loop classification is shown in Figure 5.13. The algorithm takes three inputs: the vector induction variable, the set of

```
vsum(1:64) = 0
DOVEC j = 1,64
    a(1:1+64, j) = b(j)
    vsum(1:1+64) = vsum(1:1+64) + d(1:1+64,j)
ENDVEC
c(1:64) = vsum(1:64)
```

Figure 5.12: Vector translation for outer loop

vector reference from special Non-VAR such as reduction, and the set of statements to be classified. The output is a set, V, of vector referenced objects. The algorithm starts out with a known set of vector references, namely the union of VIX and RD set. It then induces vectorization to other references knowing that:

1. A reference object(obj) must be a vector reference if $\mathcal{L}$(obj,VIX) is greater then zero

2. A statement must be a vector statement if any of its object is a vector reference.

A vector statement is one with objects in vector form. Once a vector object is identified(line 7 and 8), the remaining objects are classified as vector(line 10). These new additions to vector set V, will later convert other statements, whose members belong to V, into vector forms. Line 9 helps the algorithm converge faster but does not change the result. Once no modification is made to V after a complete pass through all the statements, the algorithm stops.

Let us apply the algorithm to example 5.11b) The outer loop is being vectorized, the computational structure for "i" is removed(by vector loop set up) and the algorithm is invoked with VIX=i, RD = empty, S = {sum = 0, j = 1, a(i,j) = b(j), sum = sum + d(i,j), j = j + 1, if (j $\leq$ 64) goto inner, c(i) = sum}. In the first pass of the loop at line-5, statements a(i,j) = b(j), sum = sum + d(i,j) and c(i) = sum are identified as vector statements since $\mathcal{L}$(a,i) = $\mathcal{L}$(d,i) = $\mathcal{L}$(c,i) = 1. After the end of the first pass of loop at line-5, V = i, a(i,j), b(j), sum, d(i,j) , S = sum = 0, j = 1, j = j + 1, if (j $\leq$ 64) goto inner. The second pass of the loop at line 5 remove "sum = 0" from S but does not add anything to V, at loop bottom V = {i, a(i,j), b(j), sum, d(i,j)} , S = {j = 1, j = j + 1, if (j $\leq$ 64) goto inner}. The algorithm then terminates, "j" will not be converted into vector form.

```
Input: VIX /* Vector induction variable */,
RD /* Vector objects resulting from special class Non-VAR */
S /* Set of statements in the loop body */
Output:V /* Set of Vector objects */
ALGORITHM
V = {VIX} union RD
change = true
while change do
    change = false
    for each statement s in S do
        Let obj = all object referenced in s
        if o ∈ obj such that
            o ∈ V ∨ ℒ(o, VIX) > 0 then
            S = S - {s}
            V = V ∪ {obj}
            change = true
        endif
    endfor
endwhile
```

Figure 5.13: Vector object classification

5.5 Summary

In this chapter we presented algorithms for implementing the results developed in earlier chapters. These algorithms aim at solving the problem of conditional privatization and outer loop vector code generation. In conditional privatization problem, we show different techniques to estimate the write set in the presence of IF-statement for containment testing. In outer loop vector code generation, we present the fundamental construct of a vector code generator and show modifications necessary to produce an outer-loop vector code generator. Results in conditional privatization are thrusted by implementation but, however, is actually a general result impacting the area of partial redundancy. The most significant aspect of pruning is that it is symbolic in nature and works in the presence of control constructs.

Chapter 6

Conclusion

In the last decade parallel vector processors(PVP) have emerged as the systems of choice to handle computationally intensive problems which require hours or days to complete. A PVP system is designed to perform many types of operations in parallel. It consists of a collection of processors each of which is capable of executing instructions independently. Many levels of parallelism are incorporated into these processors: pipelining, multi-execution of the instruction stream, chaining, low latency memory, etc. When working in concert, they are capable of delivering enormous computational power. In general, the high computational rate of a PVP system can be attributed to its ability to pipeline and to parallel process. To achieve a high computational rate, programs running on these systems must be vectorized and executed in parallel.

When there is no loop-carried dependence, a serial DO loop can be transformed into a parallel (DOALL) loop thereby decreasing its execution time. In application programs loop-carried dependencies can originate from two different sources: (1) those inherent to the problem domain and (2) those due to storage-dependence among locally computed arrays. The former inhibits DOALL transformation of the loop and (generally) cannot be overcome. However, in the latter case, DOALL transformation may still be possible by making the locally computed array private to each participating processor.

To privatize a locally computed array, three issues must be addressed: *copy-in*, *covering* and *copy-out*. Array-containment analysis focuses on the copy-in and covering problems. Most existing dependence tests permit overestimation of write sets because they are designed to solve the empty intersection problem. Since array containment analysis does not permit overestimation of write sets, most existing dependence tests are unsuitable for containment analysis. Existing methods for addressing array kill and data flow issues do not support triangular loop bounds and non-unit index stride. Array privatization methods based on integer programming techniques have been proposed, but experimental results using these methods have not been reported.

In this work, we develop a practical solution to the array containment problem. Two complementary methods, DIM and ERD, are presented. DIM supports triangular iteration spaces, coupled subscripts and symbolic loop bounds, while ERD provides union and intersection for a more restricted class of array sections. Both methods support array section containment.

The general problems of containment and copy-out in the presence of conditional statements are also addressed. Difficulty arises when, at compile time, the compiler *does not know* the value of the condition for executing an IF statement; this is the case when the condition is symbolic. This problem, to date, remains an open question. Consequently, many compiling techniques circumvent this problem by performing conservative dependence analysis. This diminishes the chance of detecting hidden concurrencies or trivial concurrencies in a program.

We propose solutions to address the symbolic conditional problems. For conditional privatization, a pruning method was developed to make use of available symbolic information. Since the symbolic condition of an IF statement must be TRUE(or FALSE) (although its value is not known at compile time), inferences can be made about the execution path involving conditionals once the THEN(or ELSE) branch is being traversed. Knowledge of an execution path enables the compiler to perform aggressive analysis by ignoring/pruning irrelevant execution paths. Thus, precise information can be gathered with pruning since interferences of false dependencies from irrelevant paths are eliminated.

The conditional copy-out problem is essentially the problem of capturing the last value (according to sequential evaluation semantics) of a privatized variable in a DOALL loop. In other words, at compile time it is difficult to determine which processor will be responsible for delivering the last value of a privatized variable. We present a generic, dynamic, solution which generates run time code to capture the value produced by that particular processor. The dynamic approach is general; however, the efficiency of the run time code generated by this approach can be improved with a certain class of monotonic write patterns which occur frequently in application programs. We refer to this improvement as the static approach.

The above results are implemented in the UNICOS environment on a Cray YMP-8 system. Experimental results lead to several interesting observations: (1) array privatization has a significant impact on loop-parallelization, (2) together, DIM and ERD can recognize a large percentage of array privatization opportunities, and (3) in cases where privatization is successful, loops are transformed to DOALL(which leads to a significant speedup). In many examples of application loops, the speedup obtained is nearly equal to the number of participating processors.

Vectorization has long been recognized as the most fundamental concept of parallel processing. It is capable of delivering enormous speedups. However, it has traditionally been limited to innermost loops, or outer loops that are made innermost by the application of loop distribution and interchange.

In this dissertation we advocate the idea of vectorizing a non-innermost loop without the application of loop distribution and interchange transformations, i.e., direct vectorization on non-innermost loops(OLV) and we develop a framework for OLV. First, from the target loop nest we identify the vector execution order that must be followed by the OLV technique. Second, we propose a set of conditions for identifying which loop (among the set of loops in a loop nest) is vectorizable in-place and which loop is not vectorizable in-place.

To perform out loop vectorization, only one loop in a loop nest can be mapped into vector form. If no loop or a single loop is vectorizable, then the choice is obvious; however, when more than one loop is vectorizable, a choice must be made. This is, in principle, an optimization problem involving many parameters. We define a metric, a weighting system, to normalize the different parameters. The vector selection process is based on this metric to identify the most beneficial (per parallelism) loop candidate for vectorization.

OLV represents a drastic change in viewpoint from ILV. The vector code generation process is different in subtle ways. This requires meticulous, thorough understanding of the ILV code generator when implementing an OLV code generator. ILV code uses sophisticated tools to map many different scalar constructs into vector forms. We present a detailed description of an innermost loop vector code generator and propose modifications necessary to create an outer loop vector code generator.

Our OLV framework is implemented in $CF90^{TM}$, the Cray Fortran 90 compiler. Experiments were performed on a Cray YMP-8 using a modified version of $CF90$ to allow compilation with or without OLV. OLV opportunities are found in 305 of 1168 vectorizable nests in the Perfect Benchmark, with OLV actually selected for 66 nests. Significant(up to 15$\times$) speedups at the subroutine level of some routines were also observed.

6.1 Summary

This dissertation aims at developing compiling technology for a parallel vector processor. Our work is in the area of loop parallelization and vectorization and our contribution is summarized as follows:

Loop Parallelization

- DIM & ERD: geometric dependence testing methods to privatize arrays.
- Conditional Privatization: a path deletion technique to analyze conditionally-defined private arrays.
- Conditional Copy-out: a transformation technique to capture the last value of a privatized variable.

Loop Vectorization

- Outer loop vectorization: a vectorization technique enabling direct vectorization of an outer loop.
- Vector selection: a metric for determining a vector loop choice.
- Outer loop vector code generator

The above compiling techniques are supported by experimental results.

6.2 Future Work

As future work, it would be interesting to develop a generic scheme which combines the benefits of DIM, ERD and the remaining capabilities discussed in section 3.2. Such a scheme would provide an even higher hit-rate for array privatization and thereby enable us to transform a larger percentage of loops into DOALL loops.

The programming model developed for an outer loop vectorization framework assumed rectangular iteration spaces. Relaxing this requirement to allow trapezoidal iteration spaces does not increase the theoretic complexity of legality testing; however, it does pose a tremendous challenge for a vector code generator. Triangular iteration spaces do appear frequently in linear algebra code and it would be fruitful to come up with an efficient code generation technique to vectorize outer loops whose iteration spaces are not rectangular.

Bibliography

[1] A. Aho, R. Sethi, and J. Ullman. *Principles of Compiler Design*. Addison-Wesley, Menlo Park, CA, 1986.

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

[3] R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, Oct. 1987.

[4] V. Balasundaram. *Interactive Parallelization of Numerical Scientific Programs.* PhD dissertation, Rice University, Apr. 1989.

[5] V. Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor. *Journal of Parallel and Distributed Computing*, 9:154–170, 1990.

[6] U. Banerjee. *Dependence analysis for supercomputing.* Parallel Processing and Fifth Generation Computing. Kluwer Academic Publishers, Norwell, Massachusetts, 1988.

[7] U. Banerjee. *Loop Transformations for Restructuring Compilers.* The Foundations, Kluwer Academic Publishers, 1993.

[8] Bradley, Hax, and Magnanti. *Applied Mathematical Programming.* Addison Wesley, Reading, MA, 1977.

[9] M. Berry et al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5-40, Fall 1989.

[10] C. Bohm and Jacopini, Flow diagrams, turing machines, and languages with on two formulation rules. *Communications of the ACM*, 19(5), May 1966.

[11] P. Bose. Heuristic Rule-Based Program Transformations for Enhanced Vectorization. *Proceedings of the International Conference on Parallel Processing*, 1988.

[12] M. Burke, R. Cytron, J. Ferrante, and W. Hsieh. Automatic generation of nested, fork-join parallelism. *The Journal of Supercomputing*, 2(3):71–88, Mar. 1989.

[13] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *SIGPLAN '86 Symposium on Compiler Construction*, pages 162–175, July 1986.

[14] D. Callahan. *A Global Approach to Detection of Parallelism*. PhD dissertation, Rice University, Apr. 1987.

[15] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *Supercomputing: 1st International Conference*, pages 138–171, 1987.

[16] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceeding of the SIGPLAN '90 Conference on Program Language Design and Implementation*, White Plains, NY, June 1990.

[17] S. Carr. *Memory-Hierarchy Management*. PhD thesis, Rice University, October 1992.

[18] R. Cytron, J. Lipkis, E. Schoenberg. A compiler-assisted approach to SPMD execution. *Supercomputing '90*, pp 398–406, 1990.

[19] R. Cytron, J. Ferrante J., and V. Sarkar. Experience using control dependence in PTRAN. In Gelernter D., Nicolau A., and Padua D., editors. *Languages and Compilers for Parallel Computing*. The MIT Press, 1990.

[20] G. Elsesser, V. Ngo, S. Bhattacharya and W.T. Tsai. Processor Pre-allocation and Load Balancing of DOALL loops *Journal of Supercomputing*, to appear, 1994.

[21] G. Elsesser, W.T. Tsai. Loop Collapse: A Systematic Approach. TR 92-61, Dept. of Computer Science, University of Minnesota, Nov 1992.

[22] P. Feautrier. Array expansion. In *1988 International Conference on Supercomputing*, pages 429–441, St. Marlo, France, 1988.

[23] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[24] T. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceeding of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[25] T. Gross and P. Steenkiste. Structured data flow analysis for arrays and its use in an optimizing compiler. *Software - Practice and Experience*, 20(2):133–155, Feb. 1990.

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

[27] L. Huelsbergen, D. Hahn, and J. Larus. Exact data dependence analysis using data access descriptors. Technical Report CS-945, Computer Sciences Department, University of Wisconsin-Madison, July 1990.

[28] K. Kennedy, K.S McKinley, and C. Tseng. Analysis and transformation in the ParaScope Editor. *Proceeding of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, July 1991.

[29] R. E. Kirk. *Experimental design : procedures for the behavioral sciences.* Brooks/Cole, Monterey, CA, 2nd edition, 1982.

[30] D. Kuck, R. Kuhn, B. Leasure, and M. Wolfe. Analysis and transformation of programs for parallel computation. *Proceeding of COMPSAC 80, the 4th International Computer Software and Applications Conference*, pages 709-715, Chicago, IL, October 1980.

[31] D. Kuck, Y. Muraoka, and S. Chen. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Transactions on Computers*, C-21(12):1293-1310, Dec. 1972.

[32] D. Kuck. The Structure of Computers and Computations. *Volume 1.*, Wiley J. and Sons, New York, NY, 1978.

[33] X. Kong, D. Klappholz, and K. Psarris. The I test: A new test for subscript data dependence. In *1990 International Conference on Parallel Processing*, pages II:204–211, 1990.

[34] R. H. Kuhn. *Optimization and Interconnection Complexity for: Parallel Processors, Single-Stage Networks and Decision Trees.* PhD dissertation, University of Illinois – Urbana-Champaign, May 1980.

[35] L. Lamport. The parallel execution of DO loops. *Communications of the ACM* 17(2):83-93, February 1974.

[36] Z. Li and P. C. Yew. Interprocedural analysis and program restructuring for parallel programs. Technical Report 720, CSRD, University of Illinois – Urbana-Champaign, Jan. 1988.

[37] Z. Li. Array Privatization: A Loop Transformation for Parallel Execution. Technical Report 92-26, University of Minnesota, April 1992.

[38] Z. Li, P.-C. Yew, and C.-Q. Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):26–34, Jan. 1990.

[39] V. Ngo, G. Elsesser, S. Bhattacharya and W.T. Tsai. Array Covering: A technique for Enabling Loop Parallelization *1992 International Conference on Parallel Processing*, August, 1992.

[40] C. D. Polychronopoulos. *Parallel Programming and Compilers*. Parallel Processing and Fifth Generation Computing. Kluwer Academic Publishers, Norwell, Massachusetts, 1988.

[41] W. Pugh. The omega test: a fast and practical integer programming algorithm dependence analysis. Technical Report UMCS-TR-91-50; CS-TR-2648, University of Maryland, Institute for Advanced Computer Science Studies and Department of Computer Science, University of Maryland, College Park, MD 20742, Apr. 1991.

[42] Cray Research. *Unicos Autotasking user's guide*. Cray Research, Inc., Eagan, MN, sn-2088 edition, 1989.

[43] Cray Research. *Cray Y-MP Hardware Reference Manual*. Cray Research, Inc., Eagan, MN, hr-4001 a edition, 1989.

[44] Cray Research, *CF90^{TM} Commands and Directives Reference Manual*. Cray Research, Inc., Eagan, MN, SR-3901 1.0, 1993.

[45] Cray Research. *UNICOS Performance Utilities Reference Manual*. Cray Research, Inc., Eagan, MN, sn-2040 edition, 1991.

[46] R. Triolet. Interprocedural analysis for program restructuring with parafrase. Technical Report 538, CSRD, University of Illinois – Urbana-Champaign, Dec. 1985.

[47] M. E. Wolfe, and M. Lam. Maximizing parallelism via loop transformations. In *Proceeding of the Third Workshop on Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[48] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD dissertation, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Oct. 1982.

[49] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. The MIT Press, Cambridge, Massachusetts, 1989.

[50] M. Wolfe. Advanced Analysis and Optimizing for Parallelism. WORKSHOP, Oregon Graduate Institute, 1993.

[51] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series. Addison-Wesley, Menlo Park, CA, 1991.