RULE-BASED PROGRAM RESTRUCTURING FOR HIGH PERFORMANCE PARALLEL PROCESSOR SYSTEMS

by

Lawrence J. Tenny

Submitted to the faculty of the Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in the Department of Computer Science

Indiana University

April 3, 1992

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

Dr. Dennis B. Gannon
(Principal Adviser)

Dr. Christopher Haynes

Dr. David B. Leake

April 3, 1992

Dr. William Wheeler

© Copyright 1992

Lawrence J. Tenny

ALL RIGHTS RESERVED

To the computer,

without whom none of this would have been possible...or necessary.

Abstract

Writing good programs for high performance parallel computers is difficult. The programmer must have a deep understanding of the underlying machine architecture. Issues such as memory hierarchy, communication topology, processor architecture, task scheduling, and processor allocation, can have a dramatic effect on a program's performance. What makes matters difficult is that, if peak performance is to be achieved, radical architectural differences require the program to be customized for each machine.

Source to source program restructuring offers the opportunity for a single incarnation of a program to be transformed at the source level so that it better matches capabilities of any given parallel computer. For serial machines, well-known compiler optimization techniques are sufficient. For parallel machines, however, the issue is not so clear. Fundamental architectural differences considerably complicate the process of improving the program-machine match. No longer does a relatively simple analysis of a program yield the most improvement. A deep understanding of the program and the architecture along with a powerful and extensible inference model is required.

In this thesis we lay the foundation for rule-based source to source program restructuring as a program optimization technique for high performance parallel processor systems. We develop rule-based methods for the basic operations required of a restructuring compiler. We develop rule-based methods for deriving program data dependencies, finding data dependence recurrences, partitioning dependence graphs into π-blocks, specifying machine properties, and selecting and planning program restructuring transformations.

Acknowledgements

This thesis represents the culmination of several years work and over 34,000 lines of code. A project of this magnitude is never solely the work one individual.

First, I would like to thank the members of my committee, Bill Wheeler, Chris Haynes, David Leake, and Dennis Gannon, for guiding this research and offering many valuable suggestions. I especially thank the chairman of my committee, Dennis Gannon. His research in parallel computing and program restructuring, inspired this effort. His perpetual faith in my abilities and his willingness help me at every turn made this work infinitely easier.

I would like to thank Charles Daffinger. Together, through many long but enjoyable hours of work, we developed several rule-based languages. One of these ultimately lead to language used in this research. He taught me OPS5, a language he learned not from a manual, but by reading the uncommented source code of a crude Common Lisp port. Tragically, a caving accident took his life just over a year before this work was completed. His friendship, tenacious dedication to programming, and unquenchable computing spirit is sorely missed.

This work would not have been possible if not for financial support I received from many generous sources. Fellowships from The Glenn A. Black Laboratory of Archaeology and the Proctor & Gamble Corporation and the support of the Department of Computer Science helped defray the cost of tuition. A dizzying array of jobs, some odd, some not so odd, with the University Computer Services allowed me to focus on research, while making enough to live.

Contents

Abstract v

Acknowledgements vi

1 Preliminaries 1

1.1 Introduction . . . 1

1.2 The Problem . . . 2

1.2.1 Parallel Organizations . . . 4

1.2.2 Levels of Parallelism . . . 6

1.2.3 Vector Execution . . . 7

1.2.4 Loop Level Parallel Execution . . . 8

1.2.5 From Algorithms to Languages to Hardware . . . 9

1.2.6 Some Examples . . . 11

1.3 The Rule-Based Approach . . . 16

1.4 Related Work . . . 16

1.5 Thesis Overview . . . 18

1.5.1 REX . . . 19

1.5.2 Overview . . . 20

2 Data Dependence Analysis 22

2.1 Introduction . . . 22

2.1.1 Overview . . . 23

2.1.2 Language Model . . . 24

2.2 Scalar Analysis . . . 24

2.2.1 Reaching Definitions 32

2.3 Subscript Analysis 34

2.3.1 Definitions 35

2.3.2 The GCD Test 37

2.4 The Data-Dependence Graph 42

2.4.1 Preliminaries 43

2.4.2 Variables, Data Flow, and Statements 45

2.4.3 Deriving the DDG 48

2.4.4 An Example 56

2.5 Summary 58

3 Recurrences **59**

3.1 Introduction 59

3.2 Transitive Closure of δ 61

3.3 Finding Recurrences 62

3.3.1 Representing Recurrences 63

3.3.2 Deriving Recurrences 65

3.4 π-Blocks: Building an Acyclic DDG 66

3.5 π-Block Order 69

3.6 Summary 70

4 Transformation Operators **72**

4.1 Introduction 72

4.2 Properties 73

4.2.1 Machine Properties 74

4.2.2 Program Properties 75

4.3 Vectorization 79

4.3.1 Finding Vectorizable Statements 79

4.3.2 Vectorizing Operators 80
4.3.3 Changing Recurrences 83
4.4 Parallelizing Loops 92
4.4.1 Localization 93
4.5 Summary 94

5 Planning **95**
5.1 Introduction 95
5.2 Planning Model 96
5.2.1 Condition Hierarchy 97
5.2.2 Property-Directed Low-Level Planning 99
5.2.3 Operator-Directed High-Level Plan Refinement 99
5.2.4 $\mathcal{M}$ Consistency 103
5.2.5 Backtracking 108
5.3 Examples 110
5.4 Summary 112

6 Organization of a Rule-Based Restructurer **113**
6.1 Introduction 113
6.2 Control 114
6.2.1 Lexicographic Conflict Resolution 114
6.2.2 Selecting Levels With Context Elements 116
6.2.3 Rule Systems as First-Class Objects 117
6.3 Support Routines 120
6.4 Summary 120

7 Thesis Summary **121**
7.1 Overview of Results 121

7.2 Future Work 123

A Rex-UPSL Syntax **125**

A.1 BNF Syntax 125

B Element Attributes **129**

B.1 Element Attribute Declarations 129

B.2 Element Attributes 129

List of Tables

1 Classification of computer architectures. 5

2 Machine property classes in $\mathcal{M}$. 74

3 Condition Hierarchy 98

4 Specialists organization of a rule-based restructurer. 118

List of Figures

1 Extremes of processor/memory organization. 6

2 Abstract syntax for the language model. 24

3 Iterative algorithm to compute reaching definitions. 33

4 A data dependence graph. 42

5 Derivation of the data dependence graph. 56

6 Dependencies before and after valid loop interchange. 84

7 Dependencies before and after invalid loop interchange. 87

Chapter 1

Preliminaries

1.1 Introduction

Writing good programs for supercomputers is difficult. The programmer must have a deep understanding of both the program and the machine architecture. Issues such as memory hierarchy, communication topology, processor architecture, task scheduling, and processor allocation can have a dramatic effect on a program's performance. To make matters worse, radical architectural differences require the programmer to customize the program for each machine in order to achieve peak performance.

Most vendors supply restructuring compilers for their machines. Restructuring compilers attempt to improve the match between the program and the unique machine architecture. These compilers can improve performance, but with limited knowledge, poorly organized heuristics, and no centralized reasoning process, they generally can not produce code that takes full advantage of the supercomputer's potential.

In this thesis we lay the foundation for rule-based source to source program restructuring as a program optimization technique for supercomputers. The transformations we use are similar to those employed by commercial restructuring compilers. What is different however, is the organization of heuristics and the use of a rule-based inference model. Using these tools, we develop methods for selecting and planning sequences of transformation operators that result in a better match between the program and the hardware.

We target our work on restructuring FORTRAN programs for parallel or vector execution on supercomputers, but there is nothing inherent in our approach that requires us to use FORTRAN. The methods described in this thesis can be used with other imperative languages. Indeed, many of the techniques developed here could be extended to more traditional program understanding problems such as serial program optimization, formal program verification, program debugging, and performance evaluation.

1.2 The Problem

Source to source program restructuring is a process in which a program is changed at the source level in a semantically invariant way so that the resulting program better matches the specific architectural characteristics of the target machine. The match improvement results a corresponding improvement in the program's performance on the target machine. For our purposes, the class of target machines is the class of parallel machines. Thus, we are concerned with transformations that improve performance on parallel machines. While the class of programming languages is the imperative languages, for practical reasons, we narrow the field to FORTRAN.

The target machine might be a real machine or an abstract machine representing a collection of machine features. In either case, the efficacy of the restructuring process is measured, in large part, by the performance enhancement obtained. With real machines, the degree of performance enhancement can be measured directly. With abstract machines, performance enhancement can usually be derived from an analytical model of the architecture under study. The analysis of real machines is often frustrated by the introduction subtle, but important, differences between the theoretic model and the real hardware.

For a serial computation model, finding and evaluating restructuring operators

that improve performance is reasonably well understood. For parallel computation models however, the issue is not so clear. Fundamental differences between memory hierarchies, communication topologies, processor power, and the availability of scheduling primitives considerably complicate the process of improving the program-machine match. No longer does a relatively simple analysis of a program yield most of the match improvements. A deep understanding of the program and the architecture is required.

A program transformation must also preserve the meaning of the original program. We must guard against introducing new program behavior when applying transformations.

A program transformation must improve performance. It is surprisingly easy to find transformations that seem reasonable, but degrade performance significantly. For the restructuring process to be useful, it must avoid applying such transformations.

These last two ideas provide us with two important requirements for the transformation process.

Validity: The transformation must preserve the semantics of the original program. The problem of validity, which is ultimately a problem of dependence analysis, becomes increasing complex as transformations are applied and parallelism is introduced.

Usefulness: For a transformation to be useful, it must improve performance. Better performance may come from many sources: vectorization, parallelization, reduced paging traffic, better cache or register utilization, reduced communication overhead, etc. Measuring the usefulness of a transformation is a difficult task. Here is where heuristics and architectural knowledge must be used by the reasoning system to guide the transformation process.

1.2.1 Parallel Organizations

One of the issues that make programming parallel computers significantly more difficult than programming serial computers is that, unlike serial computers, there is more than one fundamental parallel architecture. This means that the programmer must carefully consider *how* parallelism is to be used to best solve the problem.

Characterizations

Flynn [32] introduced a simple characterization of computer architectures based on the notion of streams [31]. A stream is a sequence of items acted on by a processor. The items can be either data or instructions.

The four combinations of single and multiple, data and instruction streams gives rise to the four architectural characterizations of computers in Table 1. The familiar serial computer is characterized by its single instruction stream and single data stream.

The family of parallel computers is characterized by either multiple data streams or multiple instruction streams, or both. The single instruction stream, multiple data stream computer (SIMD) exploits parallelism by broadcasting a single instruction or group of instructions to a set of processors. Each processor acts on a disjoint data set. Vector and array processors are typical examples of this architecture.

The multiple instruction stream, multiple data stream (MIMD) architecture is characterized by multiple processors, each executing a disjoint set of instructions on disjoint data sets. Multiprocessors like the BBN Butterfly [26] are prime examples of this type of architecture.

The last combination, multiple instruction streams acting on a single data stream (MISD), is not widely identified. However, it is reasonable to classify pipelined machines as MISD, so we label it as such in Table 1.

	Single Data Stream	Multiple Data Stream
Single Instruction Stream	**SISD** *von Neumann*	**SIMD** *data parallel*
Multiple Instruction Stream	**MISD** *pipeline*	**MIMD** *function parallel*

Table 1: Classification of computer architectures.

There are other characterizations of architectures. Most notable are those by Kuck [45] and Treleaven [66]. These are in many ways more detailed, but are much less prevalent in the literature.

Memory and Processors

In addition to the SIMD and MIMD distinctions given above, parallel machines are also characterized by memory organization. At one extreme, a set of global memories is connected to the processors via interconnection network (Figure 1a). This is sometimes called the "dance hall" configuration. At the other extreme, each processor has its own local memory (Figure 1b). This is sometimes called the "boudoir" configuration.

The dance hall configuration represents the class of uniform access, shared memory machines. Processor communication is performed directly in shared memory. The boudoir configuration represents the class of non-shared memory machines. Communication in these machines is by passing messages through the network. While the shared memory machines embody a more powerful communication paradigm, they are more difficult to implement.

There is a considerable spectrum of machines between these two extremes. Some machines with the boudoir configuration, like the BBN Butterfly, have hardware and software to support seamless shared memory access, but at considerable expense due to network latency [14]. Machines of the dance hall configuration, like the NYU Ultracomputer [38], attempt to battle the cost of network latency by associating a

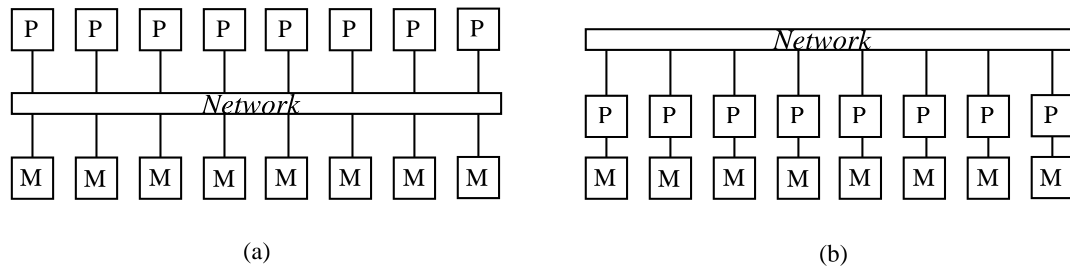

Figure 1: Extremes of processor/memory organization.

small amount of local memory with each processor.

1.2.2 Levels of Parallelism

Often referred to as the *granularity* of parallelism, there are many different possibilities for the size of parallel subtasks. For some business-oriented, commercial machines, multiple processors work on separate programs in order to increase the throughput of the system. Processors do not cooperate in the execution of a particular program. This type of parallelism is on the program level and is of very large granularity. Since we are concerned with decreasing the turnaround time of particular program, this program level parallelism is not discussed further.

Programs are often structured into one or more procedures. Assigning procedures to processors results in procedure level parallelism. The individual subtasks, procedures, represent a large number of machine instructions, hence this level represents large grain parallelism. Clearly this grain size requires a MIMD organization, but memory could be arranged either in the dance hall or the boudoir configurations, or somewhere in between.

Within a procedure, loops offer considerable opportunity for parallelism [12, 47, 49]. We will see in the discussion below that in some cases different iterations of

a loop can be distributed to different processors. This loop level parallelism offers varying grain size (depending on the number of instructions in the loop body) and is most suited for SIMD organizations, although MIMD computers can exploit this level effectively.

Finally, at the statement level, some machines have an architecture that allows a particular instruction to be executed over a collection of data. This classic SIMD organization is used for vector and array processors. This grain size is the smallest we consider, but not the smallest possible.[1]

Both the loop and statement levels of parallelism attract a considerable amount of attention in the discussion that follows. By restructuring a serial program for loop and statement level parallelism, we are essentially finding and exploiting data parallelism hidden in the original serial algorithm. More than this, however, we attempt to exploit parallelism in a way that improves performance on a particular target machine. This requires considerable knowledge of the target machine's strengths and weaknesses. Issues such as ideal grain size and processor capability play a vital role in selecting and planning a sequence of restructuring transformations.

1.2.3 Vector Execution

In a scalar computational model, an instruction consists of an operator and a small number of scalar operands. The application of the operator results in a scalar value. If the operator must be applied to several similar operands, an explicit iterative construct, such as a `DO` loop, is used.

In a vector model, an instruction consists of an operator and a small number of *sets* of operands. The operator is applied to the corresponding elements of the sets,

[1]Consider a pipelined machine in which a particular instruction and its arguments are operated on by several stages. Here the level is at the machine instruction rather than the source language statement.

producing a set of results. On an array processor, the operator is applied simultaneously to the corresponding elements. On a vector processor, the operator is applied sequentially, but through the use of pipelined hardware, the sequential application can be much faster than with the iterative process used on a conventional scalar machine.

In languages that support vector semantics, such as FORTRAN 90 [17], vector operands are specified as sections of arrays. For example, to perform an element-wise addition of the first `n` elements of arrays `a` and `b` with the result going to array `c`, a FORTRAN 90 programmer would write:

```
c(1:n) = a(1:n) + b(1:n)
```

The same operation in FORTRAN 77 requires a loop.

```
      do 10 i=1,n
       c(i) = a(i) + b(i)
  10  continue
```

The point here is not that the first version is shorter, but that for machines which have hardware to support vector operations, the first version more closely matches the machine's architecture. The better match results in better performance.

1.2.4 Loop Level Parallel Execution

Another way to achieve faster execution is to arrange for each iteration of the loop to be executed simultaneously on different processors. Some languages, like CEDAR FORTRAN [29], provide semantics for this type of execution. The previous loop would be written as,

```
    doall 10 i=1,n
     c(i) = a(i) + b(i)
 10 continue
```

Here, each iteration of the loop is scheduled for execution on the available processors. If a sufficient number of processors are available, each iteration is assigned a different processor and each proceeds more or less simultaneously.

1.2.5 From Algorithms to Languages to Hardware

There are at least three fundamental sources of parallelism: the algorithm, the programming language, and the hardware. For each of these there is an associated virtual machine with its own language and computational model [62].

For the virtual machine at the algorithm level, there is an abstract parallel computational model. This model usually conforms to either the SIMD or MIMD organization, but can contain elements of both. Communication strategies, synchronous vs. asynchronous processors, and shared vs. non-shared memory, are all issues that are settled at algorithm design time. The resulting parallel computation model and the algorithm are closely matched to provide the best possible performance.

Ultimately, the algorithm must be rendered in a programming language. The virtual machine for the language level embodies a parallel computation model that is defined by the parallel semantics of the language. Some languages, like ADA and PL/I, provide semantics for procedure level parallelism. Other languages, like FORTRAN 90, provide semantics for statement level parallelism. Concomitant with parallel semantics, each language must also provide the necessary synchronization mechanisms. Some languages, like VPC++ [35], also provide semantics for explicit domain decomposition. Whatever the extent of the semantics, the algorithm must be realized in the tools provided by the language.

Finally, at the hardware level, there is a virtual machine whose parallel computation model is embodied in silicon.[2] The virtual machine at this level encompasses all the details not considered at the higher levels. Issues such as cache size and organization, memory and network latency, and processor capabilities are at the very foundation of hardware virtual machine. Whatever the extent of the functionality presented at the language level, the program must be realized in the tools provided by the hardware.

The essential problem, and often the most serious challenge, for any parallel system is to match each succeeding level so that performance is not lost, or at least not lost to a debilitating extent.

For serial machines, this matching process is relatively easy. There are relatively minor differences between the levels in a serial system. For parallel systems, these differences can be quite dramatic. In this thesis, we are interested in improving the match between the serial language level and a parallel hardware level. This implies that the underlying algorithm and original language statements embody a serial computational model. Our challenge then is doubly hard. We are not simply improving the match between parallel models, but finding sources of parallelism that better match the underlying hardware's parallel computation model.

We view this match improvement process as one of finding and applying a series of source to source restructuring operators that transform one program form into another, semantically equivalent form, that performs better on the underlying hardware. The nature and order of the restructuring operators is of critical importance to the success of the process.

In order to preserve the original semantics of a program, considerable attention must be devoted to analyzing the data dependencies in the program — a task that in general, is undecidable. Once these dependencies have been exposed, we need to

[2] At least for the next few years.

select, plan, and apply a series of operators that preserve the original semantics while improving the program-machine match.

Dependence analysis is critically important to insure that the original semantics are preserved. We must guarantee that our optimization process does not introduce new program behavior.

1.2.6 Some Examples

The familiar algorithm for matrix multiply is usually encoded in the following way for serial computers.

```
L1: do j=1,256
L2:  do i=1,256
L3:   do k=1,256
S:     a(i,j) = a(i,j) + b(i,k) * c(k,j)
      enddo
     enddo
    enddo
```

Here we assume the matrix `a(n,n)` has been initialized to `0`. Elements of the rows and columns of matrices `b` and `c` are multiplied and the sums are accumulated in matrix `a`.

An optimizing compiler for a serial machine can directly generate good code for this loop since the accumulated sum in each element of matrix `a` can be kept in a register during iterations of the inner loop.

Let us assume that the target hardware is a vector machine capable of processing entire vectors at once. We would like to exploit this machine feature to better match the program to the hardware's capabilities and thus increase the program's performance.

One approach is to simply convert loop L_3 to vector form. However, the vector semantics of languages like FORTRAN 90 require that all operands be available simultaneously. Notice that `a(i,j)` is both used and recalculated on every iteration of the inner loop. Using terminology introduced in the next chapter, we say that there is a *loop-carried dependence* on statement S. This dependence means that the operands for the proposed vectorization of statement S are not simultaneously available, hence the dependence prohibits vectorization.

The loop-carried dependence can be modified so that vectorization is no longer inhibited. By interchanging loops L_2 and L_3, the *carrier* of the dependence is moved out one level, thus making vectorization possible.

First, the loops are interchanged,

```
L3: do k=1,256
L2:  do i=1,256
L1:   do j=1,256
S:     a(i,j) = a(i,j) + b(i,k) * c(k,j)
      enddo
     enddo
    enddo
```

then statement S is vectorized.

```
L3: do k=1,256
L2:  do i=1,256
S:    a(i,1:256) = a(i,1:256) + b(i,k) * c(k,1:256)
     enddo
    enddo
```

Although this loop would perform significantly better than the original, serial version, it can still be improved. With loop L_2 now the innermost loop, each element of `a` must be loaded and stored 256 times. By moving L_2 to the outer most position,

we can reduce the loads and stores of each element of a to just one. The reduced number of memory stores significantly improves performance on machines with cache memories.

```
L2: do i=1,256
L3:  do k=1,256
S:    a(i,1:256) = a(i,1:256) + b(i,k) * c(k,1:256)
     enddo
    enddo
```

Some machines, like the Alliant FX/8, have multiple vector processors. This architecture allows a program to use both statement level parallelism (vectorization) as well as loop level parallelism. For machines like the Alliant, the restructuring process involves the additional step of distributing different iterations of the outer loop among the available processors.

```
L1: doall i=1,256
L3:  do k=1,256
S:    a(i,1:256) = a(i,1:256) + b(i,k) * c(k,1:256)
     enddo
    enddo
```

The previous example illustrates two important points. First, the restructuring process is a *series* of transformation steps. Often, these steps involve removing or rearranging dependencies. Sometimes they involve more complex issues such as changing the grain size of a subtask to better match the optimal size of the underlying hardware. Always, they involving planning in order to meet the primary goals of validity and usefulness.

Second, the example illustrates the importance of knowning the features of the underlying hardware. In many cases, the transformation process is guided by the features of the target machine. This has lead some to term the process *feature driven* [68].

Our next example illustrates the importance of this last point. Suppose we would like to restructure the following loop for a BBN Butterfly.

```
do i=1,n
 do j=1,n
  y(i) = y(i) +  a(i,j) * x(i,j)
 enddo
enddo
```

The Butterfly is a non-uniform access, shared memory, parallel computer. The individual processors do not have vector capabilities, so our approach is to simply distribute the inner loop among the processors.

The Butterfly provides a scheduling primitive called `genoni` that executes a subroutine once for each index in a range. Restructuring the program so that the `n` iterations of the inner loop are scheduled on the available processors yields:

```
genoni(inner,n-1)
...
subroutine inner(d,i)
 i = i + 1
 do j=1,n
    y(i) = y(i) +  a(i,j) * x(i,j)
 enddo
return
end
```

Notice that each task accesses two matrices. Matrices are often quite large. The communication topology of the Butterfly and the non-uniform access, shared global memory, impose a considerable penalty for non-local memory access [14]. Ideally, we would have local copies of ith row of arrays `a` and `x` on each processor. The Butterfly has a primitive block transfer function `btransfer`[3] which efficiently moves blocks of

[3]We assume that we are using the GP1000 model. The TC2000 model emulates block transfers in software, thereby significantly reducing its usefulness here. A good example why detailed machine

memory across the communications network. For sufficiently large matrix sizes, the following version of `inner` would perform better.

```
subroutine inner(d,i)
real localx(n),locala(n)
 i = i + 1
 btransfer(x(i,*),localx,n*8)
 btransfer(a(i,*),locala,n*8)
 do j=1,n
    y(i) = y(i) + locala(j) * localx(j)
 enddo
return
end
```

Although this program is much simpler than most we might expect to encounter, we needed a considerable amount of knowledge about the machine and the program. We also reasoned with this knowledge. We concluded, for example, that `btransfer` might be useful because each processor is accessing potentially large data structures in global memory and such access is expensive. This motivated our use of `btransfer` to make local copies of the global data.

Our use of `btransfer`, however, is not free. In fact, the restructured version uses more memory and makes twice as many memory references. For sufficiently large matrices the cost of the extra local memory references is outweighed by the cost of the global memory references in the original version.

Exactly how large 'sufficiently large' is depends on a number of factors. Two of the most significant are the aggregate network bandwidth and the matrix's distribution in global memory. The restructurer must make use these and other factors in deciding the break even point. There are no hard and fast rules in this part of the domain. Heuristics must be employed to guide the restructuring process.

---

knowledge is important in program restructuring.

1.3 The Rule-Based Approach

A review of the literature on program restructuring reveals something of a potpourri of heuristics for selecting restructuring operators. Most researchers hardcode the selection heuristics; weaving the reasons for a particular operator selection deep into the fabric of their restructurers. This can make modifying the heuristics difficult and retargeting the restructurer nearly impossible. Hardcoding heuristics also makes understanding and explaining *why* a particular operator was chosen very difficult because the heuristics that guided the choice are far removed from a central organization.

The KAP [28, 43] series of retargetable restructurers, use decision tables [73, 21] to organize heuristics. While they do allow some degree of retargetability, decision tables offer only limited centralization of heuristics. Further, decision tables provide only a flat view of the heuristic space. This view makes the use of sophisticated reasoning methods much more difficult.

A rule-based approach allows for a more sophisticated, knowledge-based reasoning paradigm. Using a collection of discrete rules, a restructurer is more modular and extensible than traditional restructurers. This modularity and extensibility is crucial in developing new transformation operators, planning methods, and optimization techniques. This makes a rule-based restructurer an ideal vehicle for program restructuring research.

1.4 Related Work

The genesis of program restructuring for supercomputer was, in large part, the ILLIAC-IV project at the University of Illinois.[4] One of the early researchers in the ILLIAC-IV project was David Kuck. Kuck's research focused on programming the

[4]See [42] for an entertaining account of the turbulent history of the ILLIAC-IV.

ILLIAC-IV [48] and on uncovering parallelism in FORTRAN [47]. Others [58, 50, 24] were also at work uncovering parallelism and proposing program transformations.

Today, restructuring compilers for FORTRAN is indeed a popular research topic. The following is a brief outline of restructurers that are most visible in the literature and have influenced this research considerably.

The Parafrase project [49] at the University of Illinois is the grandfather of all restructurers. It was the first to incorporate and spawn much of the research in dependence analysis and vectorization. Building data dependence graphs, finding cycles in the graphs, and constructing what would later be called the π-block graph, were all innovations of the Parafrase project.

Allen and Kennedy's PFC (Parallel FORTRAN Converter) at Rice University [8] was originally derived from the Illinois Parafrase compiler. PFC converts FORTRAN 77 programs to a vector similar to FORTRAN 90.[5] PFC was innovative in that it implemented if-conversion [5] so that control and data dependencies could be treated uniformly. PTOOL [7], another restructurer from Rice, incorporates many of the ideas of PFC.

SUPERB (SUprenum ParallelizER Bonn) is an interactive FORTRAN parallelizer developed at the University of Bonn for the SUPRENUM project [76, 44, 37]. The target language is SUPRENUM FORTRAN similar to FORTRAN 90 with additional MIMD extensions.

KAP/205 is a commercial FORTRAN restructurer for the Cyber-205. It is a member of the family of KAP retargetable restructurers produced by a Kuck & Associates [43]. The CEDAR FORTRAN restructuring compiler built for the Ceder project at Illinois [30, 29] was also developed by Kuck & Associates.

[5]Although FORTRAN 90, then FORTRAN 8X, was not firmly specified at the time PFC was developed.

Tiny is a loop restructuring research tool developed at the Oregon Graduate Institute of Science and Technology by Wolfe [74].

Other lesser known restructuring compilers include, PAT (Parallelizing Assistant Tool) [61], ParaScope [9], PTRAN (Parallel TRANslator) [3], the Texas Instruments ASC compiler [24, 70], the Cray-1 FORTRAN compiler [40], and the Massachusetts Computer Associates Vectorizer [52, 55].

Finally, some work has focused on the rule-based approach to program restructuring. In [69, 68], Wang and Gannon propose an organizational structure for transformation heuristics and a set of transformation operators embodied in rules. Although the focus leans toward applying general techniques associated with artificial intelligence and not specifically a rule-based approach, the advantages of modularity, flexibility, and retargetability are clearly evident.

At IBM Yorktown, Bose [15, 16] has implemented a rule-based advisor called EAVE. EAVE advises programmers on the best way to write loops so that IBM's VS FORTRAN compiler will generate the most favorable code. Although EAVE does not restructure code, it does reason with similar knowledge about similar issues.

The only restructuring project that uses rule-based approach is the SAVER project under development at the University of Marburg in Germany [18]. SAVER attempts to apply both parallelization and vectorization transformations. It is not clear from the literature if SAVER will be an entirely rule-based restructurer. The preliminary reports indicate that dependence analysis is rule-based [19] and that vectorization and parallelization will be done by an "expert system."

1.5 Thesis Overview

It is important to define the scope of this thesis. The aim of our research is not to develop new transformation operators nor to discuss in depth issues in dependence

analysis. Our goal is to investigate and develop a core set of fundamental rule-based methods for program restructuring.

In this thesis, we develop rule-based methods for some of the fundamental problems in restructuring. We develop rules for deriving a data dependence graph from simple variable and flow information available from a program parse. We derive rules for finding recurrences (cycles) in the dependence graph and for finding the strongly connected components of the graph. Using these program properties and a set of machine properties, we develop rules to select and plan a sequence of transformation operators.

1.5.1 REX

Behind the examples, rules, and results discussed in this thesis is a working prototype rule-based program restructurer. The restructurer, called REX, is an experimental restructurer written in the experimental rule-based programming language, REX-UPSL. Both the language and the prototype were built for this project. The focus of the research, however, is not restructurer. REX is simply an effective tool for testing ideas. All the rules and examples in the thesis come directly from the working prototype. This is both good and bad.

By using a real programming language, our descriptions become far more precise and clear. The vague, imprecise nature of pseudo code simply does not convey the same depth of information as does an executable example. The formal nature of the language means that a rule's ancestry can often be traced directly to a theorem, definition, or lemma. This is particularly satisfying.

However, using REX means that the reader is burdened with the additional task of learning enough of the language to make sense out of the rules. Rather than devote a chapter to the syntax and semantics of REX-UPSL, we introduce the language slowly, on an as-needed basis. Hopefully, this approach serves the reader better than

a more complete introduction.

When more information is needed, the formal syntax of REX-UPSL can be found in Appendix A. Other sources of interest are the user's manual [64] and a book on a closely related language [20].

1.5.2 Overview

In Chapter 2, *Data Dependence Analysis*, we discuss the theory of dependence analysis and derive one well known test, the GCD Test, for performing subscript analysis. After the theory of data dependence analysis is introduced, we develop a small, yet powerful set of rules that derive the dependence relations and build the data dependence graph. The dependence graph and the concepts from dependence analysis are used throughout the thesis.

Chapter 3, *Recurrences*, examines the causes and consequences of cycles (recurrences) in the dependence graph. In Chapter 3, we develop a set of rules to detect recurrences in the dependence graph. In addition to finding recurrences, we present rules to partition the graph into its strongly connected components. We use this partitioned graph to determine new statement orderings after the application of a series of program transformation operators.

In Chapter 4, *Transformation Operators*, we introduce a core collection of transformation operators. Associated with each operator is a set of preconditions that must be satisfied before the operator can be applied. These preconditions are encoded in collection of rules. These rules form an essential part of the planning process.

Chapter 5, *Planning*, develops a hierarchical model for planning a sequence of operators. This is a pivotal chapter. The rationale behind the design of the rules developed in the previous chapters and the need for dependence analysis becomes even more apparent as the planning model is developed.

Chapter 6, *Organization of a Rule-Based Restructurer*, discusses the issues involved in the architecture of a restructurer. The issue of control and the ways in which control in a rule-based system can be managed are discussed in detail.

In Chapter 7, *Thesis Summary*, we review the results presented in the thesis and discuss other avenues of research that can be based on the resulted presented.

Appendix A, *Rex Syntax*, provides a formal syntactic description of REX-UPSL. This description is helpful in gaining an initial understanding the rule-based programming language used throughout the thesis.

Appendix B, *Element Attributes*, is included for reference. The reader may find it helpful to have a complete listing of the element classes and their associated attributes.

Chapter 2

Data Dependence Analysis

We begin this chapter with a brief introduction to the theory and notation of data dependence analysis. We derive one well known test, the GCD Test, that can be used to determine data dependencies in array references.

We then turn our attention to the development of rules that can derive the data dependence graph. The graph and its components are used extensively in following chapters. The presentation here serves not only to describe the dependence graph in a concise, uniform, and formal manner, but also illustrates the syntax and semantics of the rule-based language used throughout this thesis.

Much about data dependence analysis is not covered in this chapter. In particular, we do not address interprocedural analysis [67], semantic analysis [56], nor symbolic subscript analysis [39]. The goals of this chapter are to provide a foundation for our discussion of rule-based program restructuring and to present a set of rules that derive the fundamental data dependence information used in restructuring.

2.1 Introduction

The semantics of most programming languages impose a strict order on the execution of statements in the language. Except for specific transfer of control statements (eg. `goto`), the order imposed by the language is the textual order of the statements in the program. Arbitrary changes to the textual order of statements in a program

change the behavior of the program. Thus the semantics of the programming language demand a particular order of execution.

This order might seem to pose a rather serious problem for compilers that attempt to optimize serial code, since many of these optimizations result in a reordering of statements [2, 4]. Compilers that attempt to automatically vectorize or parallelize statements are not exempt from this problem. Many of the transformation operators used by these compilers change the execution order of statements.

Under certain conditions, statement ordering can be relaxed. Some statements may be executed concurrently or in a random order without changing the semantics of the resulting program. Developing rule-based methods for determining which of the statements in a given program can be reordered and what constraints exist for the set of alternate orderings is the subject of this chapter.

2.1.1 Overview

In this chapter we present a set of rules for deriving a directed graph representing all the data dependencies in a region of a program. This *data dependence graph* is an essential component of the restructuring process. Before we can discuss the graph derivation, we first introduce some notation and background.

In the next section, we discuss the notion of dependence analysis in the absence of arrays. Scalar dependence analysis serves as a vehicle for much of our introduction to the notation.

In Section 3, we include arrays in the discussion. We introduce additional notation and derive one well known test, the GCD Test, used in subscript analysis.

In the the last section we develop a rule-based representation for variables, data flow, statements, and data dependencies. Using this representation we develop a small, but powerful set of rules for deriving the data dependence graph.

program	$\longrightarrow$	procedure+
procedure	$\longrightarrow$	procedure-declaration declaration* statement+
statement	$\longrightarrow$	assignment | conditional | do-loop
assignment	$\longrightarrow$	variable = expr
conditional	$\longrightarrow$	**if** (logical-expr) **then** statement+ [**else** statement+] **endif**
do-loop	$\longrightarrow$	**do** variable = expr , expr [, expr] statement+ **enddo**

Figure 2: Abstract syntax for the language model.

2.1.2 Language Model

We use a restricted version of FORTRAN as our language model (see Figure 2). We use FORTRAN because it is the language of choice in high performance computing environments that benefit most from the restructuring processes described in this thesis.

The FORTRAN we use is restricted in the sense that our model retains some, but not all of the semantic qualities of FORTRAN. For simplicity we do not allow any of the methods for aliasing memory locations (like `equivalence` or `common` blocks), non-structured loops, loops that contain more than one induction variable, or non-linear functions in array indices. These restrictions may seem severe, but in practice, many FORTRAN programs fit this model.

The ideas presented here are not tied to the language model we have chosen. Indeed, these techniques can be adapted to virtually all imperative languages.

2.2 Scalar Analysis

Some variables, called *scalar* variables, denote individual memory locations, while others, called *array* variables, denote groups of memory locations. The importance of this distinction will become clear shortly. For now, we assume that all variables are scalar. Arrays will be handled after we have developed some basic tools.

Aside from specific control transfer actions, the semantics of our model, and indeed of most programming languages, require statements to be executed in textual order. This requirement arises directly from the often implicit requirement that the order of the side-effects to the state of the program is predictable. Consider the following statements.

S_1: z = x + y
S_2: a = z + b

If S_2 is executed before S_1, the value in the variable a would likely be different than if S_1 is executed first. It would seem that the textual order of statements in a program mandate a fixed execution order. However, consider the following, slightly different, pair of statements.

S_3: z = x + b
S_4: a = c + b

Here S_3 and S_4 may be executed in either order without changing the final values of a or z. We call this property *commutativity.*

A fundamental problem for us is to determine when two statements are commutative. Commutativity implies that parallel execution of the statements is semantically valid, assuming of course that other factors such as I/O are not involved. To see why this might be the case, consider a parallel processing model in which two or more statements are randomly distributed to two or more processors for unsynchronized execution. Due to the probabilistic nature of such a model, it would be impossible to guarantee a specific execution order while at the same time maintaining concurrent execution (ie. no synchronization). If the statements are commutative, we would require no such guarantee.

Most of the program restructuring transformations discussed in this thesis belong to a class of transformations known as reordering transformations. These transformations improve performance by reordering a group of statements without adding or removing statements. Determining commutativity for groups of statements is vital in proving the semantic invariance of a proposed reordering transformation.

In this section we describe a framework for analyzing the data dependencies in a program. Along the way we will assume that information about control is readily obtainable from a traditional control dependence analysis. For an indepth treatment of control analysis see [2] and [75]. We will take the liberty of referring to data dependence analysis as simply dependence analysis unless the reference is unclear.

We begin our discussion with a definition.

Definition 2.1 *Let P be a program in our model.*

$$S \in P \iff S \text{ is a statement in } P$$
$$VAR(S) = \{v : v \text{ is an occurrence of a variable in } S\}$$

VAR(S) is the set of all occurrences of variables in S, not simply the set of variables in S. Each occurrence of a variable in a statement is unique. We might denote a variable occurrence by its node identifier in the program's parse tree. We could introduce notation like v_i, where the subscript would be unique for each occurrence of variable v in statement S. In order to streamline matters as much as possible we introduce notation at the set level and urge the reader to keep this subtle distinction in mind when we deal at the variable level. Later, we will denote an occurrence of variable v in statement S by v_i for $1 \leq i \leq n$ where n is the number of occurrences of v in S.

Definition 2.2 *Let* P *be a program and* $S \in P$ *a statement.*

$$
\begin{aligned}
USE(S) &= \{v : v \in VAR(S) \wedge v \text{ used in } S\} \\
DEF(S) &= \{v : v \in VAR(S) \wedge v \text{ defined in } S\} \\
USE(S)_n &= \{N : N \text{ is the name of } v \wedge v \in USE(S)\} \\
DEF(S)_n &= \{N : N \text{ is the name of } v \wedge v \in DEF(S)\}
\end{aligned}
$$

Here the sets *DEF(S)* and *USE(S)* contain all the occurrences of variables defined and of variables used in *S*, respectively. The sets $DEF(S)_n$ and $USE(S)_n$ contain just the names of the variables that occur in *DEF(S)* and *USE(S)*. $DEF(S)_n$ and $USE(S)_n$ are the name projections of the corresponding sets.

We have not given a precise meaning of the term 'defined'. For now, we will avoid giving this term a precise meaning other than to suggest that it refers to variables modified in a statement. After some important notation and results are introduced, we will return to this issue.

We use the following lemma in our proof of the Commutativity Theorem. The lemma says that we have captured all the variables in *S* with the *USE* and *DEF* sets.

Lemma 2.1 *If P is a program and* $S \in P$ *a statement then,*

$$\mathrm{USE(S)_n} \cup \mathrm{DEF(S)_n} = \{n : n \text{ is the name of a variable in S}\}$$

Proof In our language model, as in most programming languages, all variables that occur in a statement occur either in a *USE* context or in a *DEF* context, or both. Hence, all variables in *S* are in either $USE(S)_n$ or $DEF(S)_n$. □

Now with the sets $USE(S)_n$ and $DEF(S)_n$ and Lemma 2.1, we are able to state the following sufficient condition for commutativity.

Theorem 2.1 (Commutativity) *Let S_1, $S_2 \in P$ be statements of program P. Then a sufficient condition for commutativity is:*

$$\begin{aligned} &(\mathrm{DEF}(\mathrm{S}_1)_\mathrm{n} \cap \mathrm{USE}(\mathrm{S}_2)_\mathrm{n}) \quad \cup \\ &(\mathrm{USE}(\mathrm{S}_1)_\mathrm{n} \cap \mathrm{DEF}(\mathrm{S}_2)_\mathrm{n}) \quad \cup \\ &(\mathrm{DEF}(\mathrm{S}_1)_\mathrm{n} \cap \mathrm{DEF}(\mathrm{S}_2)_\mathrm{n}) \quad = \emptyset \end{aligned} \tag{1}$$

Proof Our aim here is to show that if (1) holds, then the execution of S_1 followed by S_2 is semantically equivalent to the execution of S_2 followed by S_1. That is, the state of the variables mentioned in S_1 and S_2 are the same after the pair of statements have been executed in either order.

If (1) holds, then it follows that each of the clauses is $\emptyset$. We take each clause in turn.

First, $(DEF(S_1)_n \cap USE(S_2)_n) = \emptyset$ implies that no variable defined in S_1 is used in S_2. Hence, execution order will not affect members of $USE(S_2)$.

Likewise, $(USE(S_1)_n \cap DEF(S_2)_n) = \emptyset$ implies that no variable defined in S_2 is used in S_1 and again, execution order will not affect members of $USE(S_1)$.

Finally, $(DEF(S_1)_n \cap DEF(S_2)_n) = \emptyset$ implies that S_1 and S_2 do not define common variables, so the state of variables in $(DEF(S_1) \cup DEF(S_2))$ after the execution of the statements in either order is the same.

These cases account for all variables in the *USE* and *DEF* sets of each statement, and by Lemma 2.1 account for all variables in the statements. □

Notice that Theorem 2.1 does not provide us with a necessary condition for commutativity, only a sufficient one. In fact, it has been shown that the more general problem is undecidable [13].

The clauses in (1) present three different conditions that cause S_1 and S_2 to fail the sufficient condition for commutativity. If any of the clauses is not $\emptyset$, then the

statements would not meet the sufficient condition. We define a relation for each clause.

Definition 2.3 *Let S_1, $S_2 \in P$ be statements of program P such that S_1 is executed before S_2. Define,*

$$S_1\, \delta^t\, S_2 \iff DEF(S_1)_n \cap USE(S_2)_n \neq \emptyset$$
$$S_1\, \delta^a\, S_2 \iff USE(S_1)_n \cap DEF(S_2)_n \neq \emptyset$$
$$S_1\, \delta^o\, S_2 \iff DEF(S_1)_n \cap DEF(S_2)_n \neq \emptyset$$

If two statements are in the relations δ^t, δ^a, or δ^o, there is said to exist a *true* dependence, *anti* dependence, or *output* dependence, respectively, from the first statement to the second. There is an implied direction in the dependence. Expressions like, $S_i\, \delta^o\, S_j$ are read as: "There is an output dependence from S_i to S_j" or "S_j is output dependent on S_i."

We use δ without superscript annotation to denote the generic data dependence relation. That is,

$$S_i\, \delta\, S_j \iff S_i\, \delta^t\, S_j \;\vee\; S_i\, \delta^a\, S_j \;\vee\; S_i\, \delta^o\, S_j$$

Among the three relations, there is an important distinction between the true dependence δ^t, and the two relations, δ^a and δ^o. The δ^a and δ^o relations arise because our model allows variables to be re-used. If our model prohibited such re-use, as is the case in functional languages, we could reduce our discussion of dependence relations to δ^t. This also suggests dependencies that anti and output dependencies can be eliminated from programs by introducing new variables in places where variables are re-used. This and other program transformations are discussed in the following chapters.

Theorem 2.1 along with the dependence relations are a fundamental result for our work. But in order to apply Theorem 2.1 we need to examine the USE and DEF sets more closely.

For any statement $S \in P$, we might have $|DEF(S)| > 1$ or $|USE(S)| > 1$. We cannot simply refer to the statements themselves as the discrete objects of a dependence. Although we do this when we refer to a dependence from one statement to another. The object of the dependence must involve the variable which causes the dependence. Likewise, we cannot refer to a particular variable as the discrete object of a dependence because the same variable may occur more than once in a statement possibly giving rise to more than one dependence. Since the same variable my appear more than once in a statement, perhaps both in $DEF(S)$ and in $USE(S)$, we need to distinguish between occurrences of the same variable in a statement. Each of these occurrences, of course, could participate in a dependence relation.

Fortunately, our definition of $VAR(S)$ allows $USE(S)$ and $DEF(S)$ to have the property such that each occurrence of a variable is represented. The following definitions allow us to refer to individual instances of variables in a program.

Definition 2.4 *Let P be a program with only scalar variables and $S \in P$ a statement. Define,*

$$\begin{aligned}
v \equiv v' &\iff v, v' \in \left(\bigcup_{S \in P} VAR(S) \right) \wedge v \textit{ has same name and scope as } v' \\
\mathcal{D} &= \{(S, v) : v \in \text{DEF(S)} \wedge S \in P\} \\
\mathcal{U} &= \{(S, v) : v \in \text{USE(S)} \wedge S \in P\} \\
\mathcal{D}_S &= \{(s, v) : (s, v) \in \mathcal{D} \wedge s = S\} \\
\mathcal{U}_S &= \{(s, v) : (s, v) \in \mathcal{U} \wedge s = S\}
\end{aligned}$$

We say that (S, v) is an instance of a variable v *in statement* S.

The intent here is that $\mathcal{D}$ represents instances of variable definitions in P, while $\mathcal{U}$ represents instances of variable uses in P. The sets $\mathcal{D}_S$ and $\mathcal{U}_S$ represent the instances in a particular statement, S. It should be clear that for any program P and statement $S \in P$, $\mathcal{D}_S \subseteq \mathcal{D}$ and $\mathcal{U}_S \subseteq \mathcal{U}$.

We promised a more concrete definition of the term 'defined' in the remarks following Definition 2.2. We are now in a position to fulfill that promise. Consider the following statements:

S_1: z = x + y
S_2: a = z + b

Earlier we suggested that S_1 and S_2 are not commutative and Definition 2.2 along Theorem 2.1 provided us with the insight. Definition 2.3 allowed us to categorize the dependence as $S_1 \delta^t S_2$. A problem arises when another statement is introduced between S_1 and S_2.

S_1: z = x + y
S': z = c + d
S_2: a = z + b

Now does $S_1\, \delta^t\, S_2$ still hold? The answer is no. The definition of z in S_1 does not reach S_2. However, notice that both $S'\, \delta^t\, S_2$ and $S_1\, \delta^o\, S'$ hold. Hence, there still is an order imposed on the statements, but it is not quite the same as that imposed by δ. In the next chapter we consider δ^+, the transitive closure of the δ relation. For now we concentrate on the δ relation and this requires that we refine the meaning of the term 'defined' to include only the *reaching* definitions.

2.2.1 Reaching Definitions

The problem of determining the reaching definitions is vital in determining the data dependencies in a program. We briefly look at one algorithm to determine reaching definitions in order to make clear what a reaching definition is.

Definition 2.5 *Let $S \in P$ be a statement in program P.*

$$\begin{aligned}
GEN[S] &= \mathcal{D}_S \\
KILL[S] &= \bigcup_{S' \neq S} \{(S', v') : v \equiv v' \wedge v \in DEF(S)\} \\
IN[S] &= \bigcup_{S' \in P} \{(S', v) : \exists \textit{ a path from } S' \textit{ to } S \textit{ where } v \textit{ is not KILLed}\} \\
OUT[S] &= GEN[S] \cup (IN[S] - KILL[S])
\end{aligned}$$

These four sets denote the new definitions that are generated (GEN) by this statement, the definitions that mask previous definitions (KILL), the definitions that reach this statement (IN), and the definitions that survive this statement (OUT).

Figure 3 is an iterative algorithm from [2] which computes the reaching definitions, the set IN[S], for each statement. The algorithm starts by initializing the sets GEN[S], KILL[S], and OUT[S] to the set of variables defined in S, for each $S \in P$. Until no changes are made to OUT[S] for any statement $S \in P$, IN[S] is assigned the union of the OUT sets of the predecessors of S. OUT[S] is assigned GEN[S] plus the difference between the set of variables that reach S and the set of variables that are KILLed by S.

Along with reaching definitions, we need to consider uses that *live* until a particular statement. That is, we need also consider reaching uses: uses of a variable with no intervening definition until some statement $S \in P$. The algorithm for computing reaching uses is similar to the one for reaching definitions.

```
/* initialize KILL[S], GEN[S], IN[S], and PRED[S] */
 for S in P
    OUT[S] = KILL[S] = GEN[S] = the set of variables defined in S
    IN[S] = ∅
    PRED[S] = the set of predecessors of S
 end
/* iterate through P until no changes are made */
 change = TRUE
 while(change)
    change = FALSE
    for S in P
     IN[S] = ⋃_{S'∈PRED[S]} OUT[S']
     OLD = OUT[S]
     OUT[S] = GEN[S] ∪ (IN[S] - KILL[S])
     if OUT[S] != OLD then change = TRUE
    end
 end
```

Figure 3: Iterative algorithm to compute reaching definitions.

Now that reaching definitions and reaching uses have been introduced, we can modify our previous definition of the δ relations as follows.

Definition 2.6 *Let S_1, $S_2 \in P$ be statements of program P such that S_1 is executed before S_2. Let REACHD(S_1,S_2) be the set of instances of variable definitions that reach statement S_2 from S_1 with no intervening definition. Let $REACHD(S_1, S_2)_n$ be the name projection of REACHD(S_1,S_2). Let REACHU(S_1,S_2) be the set of instances of variable uses that reach statement S_2 from S_1 with no intervening definition. Let $REACHU(S_1, S_2)_n$ be the name projection of REACHU(S_1,S_2).*

$$S_1\, \delta^t\, S_2 \iff (\mathrm{DEF(S_1)_n} \cap \mathrm{REACHD(S_1, S_2)_n}) \,\cap\, \mathrm{USE(S_2)_n} \neq \emptyset$$

$$S_1\, \delta^a\, S_2 \iff (\mathrm{USE(S_1)_n} \cap \mathrm{REACHU(S_1, S_2)_n}) \,\cap\, \mathrm{DEF(S_2)_n} \neq \emptyset$$

$$S_1\, \delta^o\, S_2 \iff (\mathrm{DEF(S_1)_n} \cap \mathrm{REACHD(S_1, S_2)_n}) \,\cap\, \mathrm{DEF(S_2)_n} \neq \emptyset$$

This definition of the δ relations is more precise, yet somewhat more complex than our previous definition. Throughout the remaining part of this thesis, whenever we refer to a δ relation, we implicitly refer to this definition.

2.3 Subscript Analysis

In the previous section we restricted our attention to scalar variables. Here we shift the focus to dependence analysis in the presence of arrays, especially in the context of loops. Arrays, often called subscripted variables or vectors, are used extensively in loops. Loops offer a great potential source for speedup in parallel systems [47, 12, 49]. However, vectors used in the context of loops complicate the analysis considerably [71, 72, 75]. Most of the restructuring transformations discussed in the following chapters are applied to loops and thus rely extensively on the information obtained from the dependence analysis of arrays in loops.

In this section we introduce some additional dependence notation along with a few new concepts. We derive one well known dependence test, the GCD Test, that can be used to disprove the existence of a dependence. The general problem of proving the existence of a dependence based on arbitrary subscript expressions is undecidable.

Consider the following loop.

```
do i = 1,10
 S1: z(i) = 2 * x(i)
 S2: y(i) = z(i) + 1
enddo
```

Intuitively we see that $S_1\ \delta^t\ S_2$, and indeed if we simply assumed that z was a scalar variable using the results of the previous section we would conclude that S_2 is dependent on S_1. However, consider following slightly different loop.

```
do i = 1,10
 S1: z(i*2) = 2 * x(i)
 S2: y(i) = z(i*2+1) + 1
enddo
```

Here we would again conclude $S_1 \delta^t S_2$, but we see that S_1 defines the even elements of z while S_2 uses the odd elements of z. Our scalar analysis fails because we are forced to treat the entire array z as a single variable.

What we need to look at is not simply which variables are used or defined, an approach that was sufficient for scalars, but which positions within the arrays are used or defined. Since these positions are referenced by subscript expressions, the fundamental problem is to determine when these subscript expressions refer to the same element in an array.

2.3.1 Definitions

In our language model as in most programming languages, loops may be nested to virtually any depth. In practice, nesting to several levels is quite common so our approach should be general enough to handle loop nesting to an arbitrary depth.

Definition 2.7 *Let $L_1, \ldots, L_n$ be loops nested to depth n in program P. We define an iteration vector for $L_1, \ldots, L_n$ to be $\mathbf{i} = (i_1, \ldots, i_n)$ where each i_j in $1 \leq j \leq n$ is the value of the induction variable of loop L_j for some iteration of the loop.*

We say that $\{\mathbf{i} : \mathbf{i}$ is a vector in $L_1, \ldots, L_n\}$ is the iteration space of the loop $L_1, \ldots, L_n$.

An iteration vector describes the state of each induction variable in any particular iteration of the loop. There is a different iteration vector for each possible value of each induction variable. For example, in the following loop we have $\{\mathbf{i}\} = 10 \times 20 \times 5$.

```
L1: do i=1,10
L2:  do j=1,20
L3:   do k=1,5
...
```

Loops need not be, and often are not, perfectly nested. We might have a nesting similar to the loop below. Here the length of the iteration vector for L_1 is 1, but the length of the iteration vector for L_1, L_2 and L_1, L_3 is 2.

```
L1: do i=1,25
L2:  do j=1,4
S1:   a(j,i) = c * b(i,j)
     enddo
     c = e(i)
L3:  do j=1,4
      a(j,i) = c * d(i,j)
     enddo
    enddo
```

The association of an iteration vector with a statement is written as $S(\mathbf{i})$ and denotes a particular instance of the statement in an iteration of a loop. For example, $S_1((1,3))$ is the statement instance: `a(3,1) = c * b(1,3)`.

As in the previous section, the central problem is to determine when two statements are in a δ relation. We say that two statements $S_1, S_2 \in P$ are dependent if and only if there exists some pair of iterations vectors $(\mathbf{i}, \mathbf{i}')$ such that $S_1(\mathbf{i})\ \delta^t\ S_2(\mathbf{i}')$, $S_1(\mathbf{i})\ \delta^a\ S_2(\mathbf{i}')$, or $S_1(\mathbf{i})\ \delta^o\ S_2(\mathbf{i}')$ holds.

Definition 2.8 *Let $S_j, S_k \in P$ be two statements in loop $L_1, \ldots, L_n$ and let $S_j(\mathbf{i})$, $S_k(\mathbf{i}')$ be a pair of statement instances where $(\mathbf{i}, \mathbf{i}')$ is in the iteration space of the loop. Define,*

$$\mu_j = i'_j - i_j$$

$$
\begin{aligned}
\theta_j &= \begin{cases} > & \mu_j < 0 \\ = & \mu_j = 0 \\ < & \mu_j > 0 \end{cases} \\
\boldsymbol{\mu} &= (\mu_1, \ldots, \mu_m), \text{ where } m = \min(|\mathbf{i}|, |\mathbf{i}'|) \\
\boldsymbol{\theta} &= (\theta_1, \ldots, \theta_m), \text{ where } m = \min(|\mathbf{i}|, |\mathbf{i}'|)
\end{aligned}
$$

We call $\boldsymbol{\mu}$ *the* **distance** *vector and* $\boldsymbol{\theta}$ *the* **direction** *vector between* $\mathbf{i}$ *and* $\mathbf{i}'$*.*

Definition 2.9 *Let* $*$ *refer to an arbitrary direction* $\theta \in \{<, >, =\}$*. We define the set of plausible direction vectors,* $\psi(S_j, S_k)$*, between two statements* $S_j, S_k \in P$ *where* S_j *occurs textually before* S_k *in* P *as,*

$$\psi(S_j, S_k) = \{=, \ldots\} \cup \{=, \ldots, <, *, \ldots\} \cup \{<, *, \ldots\}$$

We will use the direction vector in stating the GCD Test and in deriving the dependence graph. The direction vector is useful in categorizing a dependence between two statements. If the distance vector between two statment instances $S_j(i)$ and $S_k(i')$ is non-zero and $S_j(i)\ \delta\ S_k(i')$ holds, then we say that the dependence is *loop-carried.* We denote a loop-carried dependence by δ_c where c is the distance between the iteration vectors. If the distance vector is zero, we say that the dependence is *loop-independent.*

2.3.2 The GCD Test

Our present challenge is to determine if two statements are dependent given the possible set of iteration vectors and the subscript expressions contained in the array references of the statements. The following loop illustrates the problem.

```
do i=1,10
 S1: a(8*i-2) = d
 S2: c = a(2*i)
enddo
```

We would like to know if there exists iteration vectors $(\mathbf{i}, \mathbf{i}')$ such that $S_1(\mathbf{i})\,\delta\,S_2(\mathbf{i}')$ holds. In other words, we would like to know it there is a solution[1] to the following dependence equation, $8x - 2y = 2$ where $1 \leq x, y \leq 10$.

Recall that we limit subscript expressions to simple linear functions. In general, we can write the dependence equation as,

$$a_0 + \sum_{1 \leq j \leq n} a_j i_j = b_0 + \sum_{1 \leq j \leq n} b_j i'_j$$

which we may write in a more familiar form as,

$$\sum_{1 \leq j \leq n} a_j i_j - \sum_{1 \leq j \leq n} b_j i'_j = (b_0 - a_0)$$

which is a linear diophantine equation.

If there is a solution within the bounds of our induction variables then we know that a dependence exists. On the other hand, if we can prove that no solution exists, then no dependence exists. In order to develop this idea further, we need the following results.

Lemma 2.2 *Let*

$$\sum_{1 \leq i \leq n} a_i x_i = c$$

---

[1] Let $\mathbf{i} = (2)$ and $\mathbf{i}' = (7)$.

be a linear diophantine equation and $g = \gcd(a_1, \ldots, a_n)$*, then*

$$\exists v_1, \ldots, v_n : \sum_{1 \leq i \leq n} a_i v_i = g$$

Proof Let C = $\{\sum_{1 \leq i \leq n} a_i v_i : v_i \in \mathcal{Z}\}$ and $c' = \min\{c \in C \wedge c \neq 0\}$. First, we claim that c' divides each a_i, $1 \leq i \leq n$. Assume that c' does not divide a_j for some j in $1 \leq j \leq n$, then

$$\begin{aligned} a_j &= c'q + r \\ r &= a_j - c'q \\ &= a_j - (\sum_{1 \leq i \leq n} a_i v_i)q \\ &= a_j(1 - v_j)q + \sum_{\substack{i \neq j \\ 1 \leq i \leq n}} -a_i v_i q \end{aligned}$$

Hence $r \in C$, but this contradicts the definition of c'. Since a_j was chosen arbitrarily, it follows that c' divides each a_i.

Next, we claim that $g = \gcd(a_1, \ldots, a_n) = c'$. If $g < c'$ then c' is a greater common divisor of $a_1, \ldots, a_n$ which contradicts the definition of g.

If $g > c'$ then g does not divide c'. But, from the previous claim, c' divides $a_1, \ldots, a_n$ so g must divide c'. Hence, $g = c'$ and

$$\exists v_1, \ldots, v_n : \sum_{1 \leq i \leq n} a_i v_i = \gcd(a_1, \ldots, a_n) = g$$

□

Theorem 2.2 *Let*

$$\sum_{1 \le j \le n} a_j x_j = c$$

be a linear diophantine equation, and $g = \gcd(a_1, \ldots, a_n)$*. The equation has a solution if and only if* g *divides* c*.*

Proof If g divides c then $gk = c$ and from Lemma 2.2 there exists $v_1, \ldots, v_n$ such that $\sum_{1 \le i \le n} a_i v_i = g$. Hence,

$$gk = c = k \sum_{1 \le i \le n} a_i v_i$$

and the solutions are $(v_1 c/g, \ldots, v_n c/g)$.

If the equation has a solution $(u_1, \ldots, u_n)$ and g divides each a_i $1 \le i \le n$, then clearly g divides c. □

Corollary 2.1 *Let*

$$\sum_{1 \le j \le n} a_j x_j = c$$

be a linear diophantine dependence equation associated with $S_m, S_n \in P$*. If the equation has no solution then* $S_m \ \delta \ S_n$ *does not hold.*

Theorem 2.2 provides us with a simple way to prove that no solution exists, and by Corollary 2.1 that no dependence exists. But when $\gcd(a_1, \ldots, a_n)$ does divide c, the theorem does not tell us where the solution is. In particular, it does not tell us if the solution exists within the iteration space of the loop. This problem is rather fundamental when dealing with arrays. The iteration space of a loop is not always known at compile time because the loop bounds may depend on values computed at run time. Using only the corollary, we are forced to take the conservative approach and assume a dependence exists whenever the equation has a solution, even though the solution may not fall within the iteration space of the loop.

Although the following theorem does not tell us if a solution exists within the iteration space, the GCD Test can indicate if the dependence is loop-carried or loop-independent.

Theorem 2.3 (GCD Test) *Let $a_1, \ldots, a_m$ and $b_1, \ldots, b_n$ be the coefficients of a dependence equation, $\boldsymbol{\theta}$ be a direction vector between two statement instances $S(\mathbf{i})$ and $S'(\mathbf{i}')$, and $g = \gcd(\{a_j - b_j : \theta_j = \text{`='}\}, \{a_j : \theta_j \neq \text{`='}\}, \{b_j : \theta_j \neq \text{`='}\})$.*

If $S\ \delta\ S'$, then g divides $(b_0 - a_0)$.

Proof We may write the dependence equation as,

$$\sum_{1 \leq j \leq n} \{(a_j - b_j)i_j : \theta_j = \text{`='}\} + \sum_{1 \leq j \leq n} \{(a_j i_j : \theta_j \neq \text{`='}\} - \sum_{1 \leq j \leq n} \{(b_j i'_j : \theta_j \neq \text{`='}\} = b_0 - a_0 \quad (2)$$

Since (2) is a linear diophantine dependence equation associated with S and S', if $S(\mathbf{i})\ \delta\ S'(\mathbf{i}')$ then by Corollary 2.1, g divides $(b_0 - a_0)$. □

To illustrate how Theorem 2.3 might be used to distinguish between loop-carried and loop-independent dependencies, consider the following loop.

```
L1: do i=1,10
L2:  do j=1,10
S1:   a(2*i+3*j+2) = d
S2:   c = a(5*i+9*j+4)
     enddo
    enddo
```

The dependence equation is $2 + 2x_1 + 3x_2 = 4 + 5y_1 + 9y_2$. Which by Theorem 2.2 has a solution since $\gcd(5, 9, -2, -3) = 1$ which divides 2.

If we choose $\boldsymbol{\theta} = (=, =)$ we have $\gcd(5 - 2, 9 - 3) = \gcd(3, 6) = 3$ which does not divide 2. By Theorem 2.3 there is no dependence between $S_1(\mathbf{i}), S_2(\mathbf{i})$ and if a dependence exists within the iteration space it is a loop-carried dependence.

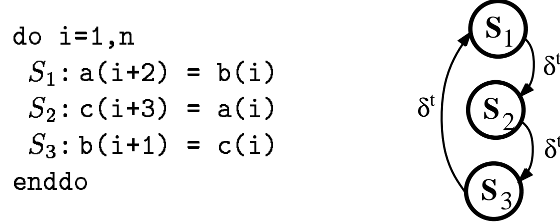

Figure 4: A data dependence graph.

Another dependence test, the *Separability Test* [71], is based on the explicit representation of the solution space of the dependence equation. The Separability Test can be applied only if the dependence equation has exactly one induction variable. However, the test yields both a necessary and sufficient condition for dependence.

Finally, the *Banerjee Test* [10] can be used to determine of a solution exists within the iteration space but can not be used to determine if the solution is integer or real. Since we require that the solutions be integer, the Banerjee Test can only be used to disprove a dependence.

2.4 The Data-Dependence Graph

A data dependence graph (DDG) [46] is a digraph $G = (V, E)$ where $V = \{S_i \,|\, S_i \in P\}$ and $E = \{(S_i, S_j) \,|\, S_i \,\delta\, S_j\}$. Edge direction in G corresponds to the implied direction of the dependence. We label each edge with the corresponding dependence relation: δ^t, δ^a, or δ^o (see Figure 4).

The DDG serves as the abstract representation of the data dependencies in the program. The relations represented in the DDG namely, δ^t, δ^a, and δ^o, characterize the cause of non-commutativity. As we will see in the chapters that follow, the problem of finding the correct sequence of program transformations that result in some desired goal is essentially one of finding isomorphisms to a subgraph of the DDG. In this section we develop a concise set of rules for deriving the DDG from

simple variable, data flow, and statement information.

This approach is unique in many respects. On the practical side, it allows us to treat changes to the DDG in a uniform, well defined manner. This uniformity greatly simplifies the otherwise complex program dependence information and makes updating the dependences, especially after a series of program transformations, not only possible, but relatively easy (see Section 5.2.4).

A more fundamentally unique and advantageous feature of this approach is that it provides a clear, syntactic, and even executable definition of precisely what the DDG represents.

Finally, an equally important advantage is that the representation presented here allows us to exploit the powerful pattern matching facilities inherent in rule-based languages to find the graph isomorphisms that are the essence of the restructuring process.

2.4.1 Preliminaries

The rules presented in this section and in the following chapters are written in an experimental programming language developed for this project. The language, called Rex-UPSL [64], supports both a traditional SCHEME [59] environment as well as a tuple space similar to that of a relational database [27]. The tuple space is called *memory* and is denoted by $\mathcal{M}$. The individual tuples in $\mathcal{M}$, called *memory elements*, are denoted by $\mathcal{M}_e$.

Definition 2.10 *A memory element $\mathcal{M}_e$, is an ordered tuple of the form*

$$(class\ obj\ obj \ldots)$$

where class is a Scheme symbol and obj is a Scheme object.

Memory, $\mathcal{M}$, is the collection of all memory elements.

With each position in the tuple we associate an attribute name. This allows symbolic access to objects within a tuple.

Definition 2.11 *An attribute name within a memory element is any symbol preceded by the ^ special character.*

Attribute names are associated with class names in declaration statements (see Appendix B). For example,

```
(declare-attribute box color width height depth)
```

associates the attributes `color`, `width`, `height`, and `depth` with the class name `box`. The following memory element describes a particular instance of `box`.

```
(box ^color red ^width 12 ^height 10 ^depth 8)
```

A particular rule in a Rex-UPSL program is applicable when the preconditions specified in the rule are satisfied by one or more tuples, $\mathcal{M}_e \in \mathcal{M}$.

Definition 2.12 *A rule is an expression of the following form.*

$$\texttt{(rule } \textit{name } \underbrace{\textit{pattern pattern}\ldots}_{LHS} \texttt{-->} \underbrace{\textit{expression}}_{RHS}\texttt{)}$$

The left hand side (LHS) is a conjunction of patterns, called condition elements. The right hand side (RHS) is an arbitrary Rex-UPSL expression.

A rule consists of two parts: a left hand side and a right hand side. The left hand side is a conjunction of patterns similar to a prototypical $\mathcal{M}_e$, but may contain

variables or relational operators. The right hand side is an expression that is a candidate for evaluation when the left hand side patterns are consistently matched by some subset of $\mathcal{M}$.

It is important to note here that rules in Rex-UPSL are declarative objects in much the same sense as clauses in Prolog. Rules do not examine $\mathcal{M}$ when encountered by the Rex-UPSL interpreter. Rather, the set of satisfied rules is determined after each change to $\mathcal{M}$. This set is constructed using an efficient, state saving, many pattern, many object matching algorithm [34].

The syntax and semantics of our rule-based language will be illustrated here by way of example and discussion. A formal syntactic description is presented in Appendix A. See [64] for a full discussion of the language.

2.4.2 Variables, Data Flow, and Statements

We begin our discussion of the DDG derivation rules by developing a representation in $\mathcal{M}$ for variables, data flow, and statements in a region R of program P. We use the following definitions.

Recall that a variable instance is the pair (S_j, v_i) where $S_j \in P$ and v_i is an occurrence of variable v in S_j.

Definition 2.13 *For each variable instance (S_j, v_i), let $t_{(S_j,v_i)}$ be a unique positive integer. We will use $t_{(S_j,v_i)}$ as a tag to refer to the instance (S_j, v_i).*

Definition 2.14 *Let t_v and t_{S_j} be positive integers such that,*

$$\forall_{v,v'}\ v \equiv v' \leftrightarrow t_v = t_{v'}$$

and,

$$\forall_{k \neq j}\ t_{S_j} \neq t_{S_k}$$

We will use t_v and t_{S_j} as tags to refer to variables and statements.

Definition 2.15 *For each variable v,*

let v_n be the name of variable v, and,

$$\text{let } v_r \text{ be } \begin{cases} \texttt{VAR_REF} & \text{if } v \text{ is a scalar variable} \\ \texttt{ARRAY_REF} & \text{if } v \text{ is an array variable} \end{cases}$$

Variables

The set of variable instances referenced in a region of a program is represented in $\mathcal{M}$ by elements of class `var`. Five attributes are associated with each variable instance.

- `stmt` is the statement's tag, t_S.
- `access` is the symbol `def` if the variable instance represents a definition of the variable, or is the symbol `use` if the instance represents a use of the variable.
- `instance` is the tag, $t_{(S_j,v_i)}$, of the variable instance.
- `var` is the variable's name, v_n.
- `ref` is `ARRAY_REF` if this is an array reference or `VAR_REF` if it is a scalar reference.

For each instance of each variable v accessed in a use context in each statement S_j in region R of program P, $\mathcal{M}$ contains:

$$(\texttt{var}\ \ \hat{}\texttt{var}\ v_n\ \ \hat{}\texttt{access}\ \texttt{use}\ \ \hat{}\texttt{instance}\ t_{(S_j,v_i)}\ \ \hat{}\texttt{stmt}\ t_{S_j}\ \ \hat{}\texttt{ref}\ v_r\ )$$

Likewise, for each instance of each variable v accessed in a definition context in each statement S_j in region R of program P, $\mathcal{M}$ contains:

$$(\texttt{var}\ \ \hat{}\texttt{var}\ v_n\ \ \hat{}\texttt{access}\ \texttt{def}\ \ \hat{}\texttt{instance}\ t_{(S_j,v_i)}\ \ \hat{}\texttt{stmt}\ t_{S_j}\ \ \hat{}\texttt{ref}\ v_r\ )$$

By Lemma 2.1, all variable instances in region R are represented in $\mathcal{M}$.

Data Flow

The reaching definitions and reaching uses discussed in Section 2.2.1 are represented in $\mathcal{M}$ by the class `flow`. Four attributes are associated with the class `flow`.

- `var` is the variable's name.
- `from` is the tag, t_{S_i}, of the initial flow point.
- `to` is the tag, t_{S_j}, of the terminal flow point.
- `flow-type` is `def-use` if the definition of v in S_i reaches a use of v in S_j, `use-def` if a use of v in S_i reaches a definition of v in S_j, or `def-def` if a definition of v in S_i reaches a definition of v in S_j.

The reaching definition of variable `a` in,

```
S1: a = 89
S2: b = c**2
S3: c = a*c
```

is represented in $\mathcal{M}$ as,

(`flow ^var a ^from` t_{S_1} `^to` t_{S_3} `^flow-type def-use`)

Statements

Statements are represented in $\mathcal{M}$ by the class `statement`. The following five attributes are used.

- `class` is the classification of the statement, eg. ASSIGN for assignment statements.

- `stmt` is the statement's tag, t_S.
- `depth` is the nesting depth of this statement within loops, starting from 0.
- `control` is the tag of the control parent for this statement.
- `order` is an integer indicating relative textual ordering of this statement.

The `order` attribute indicates the relative textual ordering of statements. Textual ordering is a weaker notion than execution ordering. Execution ordering is essential in determining the reaching definitions and reaching uses of a variable. We will see in Section 2.4.3 that textual order is sufficient to distinguish between loop-independent and loop-carried dependencies.

In the next section we show how the DDG can be derived from this representational set. This is not the only representational set, nor is it the smallest. There are disadvantages in using this set. Most notably, we sacrifice the ability to handle variable aliases by not explicitly representing execution paths. The set we have chosen does allow us to use a much smaller collection of rules to derive the DDG. Essentially, we have pushed some of the complexity of the derivation into the builtin flow analysis (see Figure 3) at the price of reducing the number of things we can reason about.

2.4.3 Deriving the DDG

With a uniform representation in $\mathcal{M}$ for the variables, data flow, and statements, in this section we present six rules for deriving the DDG.

Representing DDG Edges

Edges in the DDG are the dependencies in region R. Each edge is represented by the class `dependence` and is annotated with the following attributes.

- `var` is the variable name.
- `from` is the statement tag of initial point.
- `to` is the statement tag of terminal point.
- `type` is one of: `true`, `anti`, or `output`.
- `ref` is either `ARRAY_REF` or `VAR_REF`.
- `extent` is either `loop-independent` or `loop-carried`.
- `carrier` is the statement tag of the loop that carries the dependence.
- `vfrom` is the variable instance tag of the initial point.
- `vto` is the variable instance tag of the terminal point.

Scalar Analysis

We begin with three rules that build the initial true, output, and anti dependence edges. Recall that for a true dependence between statement S_i and S_j, there must be a definition of some variable v in S_i and a use of v in statement S_j with no intervening definition of v. That is, the definition of v in S_i reaches the use in S_j.

In Rex-UPSL, these preconditions are expressed in the following rule.

Rule 2.1

```
(rule true
  (var ^access def ^var <v> ^instance <vt1> ^stmt <st1> ^ref <r>)
  (var ^access use ^var <v> ^instance <vt2> ^stmt <st2> ^ref <r>)
  (flow ^flow-type def-use ^var <v> ^from <st1> ^to <st2>)
-->
  (make dependence ^var <v> ^type true ^from <st1> ^to <st2>
     ^vfrom <vt1> ^vto <vt2> ^ref <r> ^extent nil)
  (add-directions <vt1> <vt2> <st1> <st2>))
```

First, a note on syntax. The condition elements on the left hand side are the preconditions for the application of the rule. The condition elements look like memory elements except for the brackets < > surrounding some of the identifiers. These brackets denote variables in the condition element. Variables mentioned more than once on the left hand side must match the same value. When the rule becomes active, these values are available for use in the right hand side expression.

Here the left hand side has three condition elements. The first and second match any pair of instances of a particular variable in which the variable occurs in a def context and a use context, respectively. The third condition element matches any of the flow memory elements involving a definition of the variable in one statement <st1>, followed by a use of the variable in another statement <st2>, with no intermediate definition.

Because multiple occurrences of a condition element variable must match the same value, the first two condition elements match instances of the same variable. The integer tags representing the instances are bound to the condition element variables, <vt1> and <vt2>. Likewise, because the condition element variable <v> must be bound consistently, the third condition element restricts which memory elements the first and second condition elements can match. These involve a definition of <v> in statement <st1> followed by a use of <v> in statement <st2> with no intervening definition along any execution path from <st1> to <st2>.

On the right hand side, the make special form builds a memory element from its arguments. The first argument is the element class. The remaining arguments are the attribute-value pairs for the element.

The function add-direction-vectors will be discussed shortly.

The rules for output and anti dependence edges are similar.

Rule 2.2

```
(rule output
  (var ^access def ^var <v> ^instance <vt1> ^stmt <st1> ^ref <r>)
  (var ^access def ^var <v> ^instance <vt2> ^stmt <st2> ^ref <r>)
  (flow ^flow-type def-def ^var <v> ^from <st1> ^to <st2>)
-->
  (make dependence ^var <v> ^type output ^from <st1> ^to <st2>
     ^vfrom <vt1> ^vto <vt2> ^ref <r> ^extent nil)
  (add-directions <vt1> <vt2> <st1> <st2>))
```

Rule 2.3

```
(rule anti
  (var ^access use ^var <v> ^instance <vt1> ^stmt <st1> ^ref <r>)
  (var ^access def ^var <v> ^instance <vt2> ^stmt <st2> ^ref <r>)
  (flow ^flow-type use-def ^var <v> ^from <st1> ^to <st2>)
-->
  (make dependence ^var <v> ^type anti ^from <st1> ^to <st2>
     ^vfrom <vt1> ^vto <vt2> ^ref <r> ^extent nil)
  (add-directions <vt1> <vt2> <st1> <st2>))
```

The following example illustrates how these rules work. Suppose we have two statements with tags t_{S_1} and t_{S_2} that reference variable count. Statement t_{S_2} assigns to count some value (i.e. count $\in$ DEF(S_2)), while t_{S_1} reads from count (i.e. count $\in$ USE(S_1)). $\mathcal{M}$ would then contain the following three memory elements.

```
(var ^var count ^access use ^instance t_(S1,count) ^stmt t_S1 ^ref VAR_REF)
(var ^var count ^access def ^instance t_(S2,count) ^stmt t_S2 ^ref VAR_REF)
(flow ^var count ^from t_S1 ^to t_S2 ^flow-type use-def)
```

The first two preconditions of the anti rule are satisfied by the two var elements. These represent the two instances of the variable count that occur in statements t_{S_1} and t_{S_2}. The condition element variables <vt1> and <vt2> are bound to the tags for use instance and the defining instance, respectively.

The third precondition is satisfied by the `flow` element. This element represents a reaching use of t_{count} from statement S_1 to statement S_2. The resulting `dependence` edge is represented by the memory element,

```
(dependence ^var count ^type anti ^from t_S1 ^to t_S2
     ^vfrom t_(S1,count) ^vto t_(S2,count) ^ref VAR_REF ^extent nil)
```

In addition to creating the dependence edge, each of the rules invokes a function called `add-direction-vectors`. This builtin function applies the GCD Test to the variable instances. The resulting direction vectors are added to $\mathcal{M}$. We will use these direction vectors in subscript analysis.

Direction vectors are represented by the element class `direction`. The following attributes are used.

- `dir` is a direction, `<`, `>`, or `=`.
- `dim` is the dimension described by this element.
- `depth` is the depth described by this element.
- `var-from` is the initial point of the dependence.
- `var-to` is the terminal point of the dependence.

Each dimension in a multi-dimensional array gives rise to a set of direction vectors. Each dimension/depth pair is represented by a memory element.

Subscripted Variables

Because they do not distinguish between loop-carried and loop-independent dependencies, the previous set of rules is not sufficient. In the chapters that follow, we will

make considerable use of the `extent` and `carrier` attributes. So it is vital that we have rules to fill in these attributes.

More importantly, the rules presented so far treat all variables as scalars, thus do not consider the effects of subscript expressions in array references. The subscript analysis, performed by the builtin GCD Test, is represented by the `direction` elements in memory. Here, we use the direction vector information to fill in both the `extent` and `carrier` attributes as well as to filter out dependencies that are not supported by the direction vectors.

The following rule finds dependencies that are loop-independent. It is more complex than the previous rules.

Rule 2.4

```
(rule loop-independent
  {<ce> (dependence ^from <st1> ^to <st2>
     ^vfrom <vt1> ^vto <vt2> ^extent nil)}
  (direction ^dir '= ^var-from <vt1> ^var-to <vt2>)
  -(direction ^dir <> '= ^var-from <vt1> ^var-to <vt2>)
  (statement ^stmt <st1> ^order <pos>)
  (statement ^stmt <st2> ^order {> <pos>})
-->
  (modify <ce> ^extent loop-independent ^carrier 0))
```

The variable `<ce>` is associated with the first condition element on the left hand side. Because variables can appear on either end of a condition element, the brackets force the correct association. The variable `<ce>` is bound to the memory element that successfully matched the condition element.[2]

The negation unary operator, `-`, appears with the third condition element. This causes rule satisfaction to *fail* if there exists elements in memory that match the corresponding condition element. For Rule 2.4, the third condition element requires that no direction different from `=` exists for this dependence. The second condition

[2]Memory elements are represented externally as REX-UPSL vectors. Memory elements are first-class objects.

element requires that there is at least one = direction in the direction vector. Together, condition elements two and three guarantee that the direction vector has only = members.

Finally, the right hand side uses the `modify` special form to change the memory element representing the dependence edge. The `extent` and `carrier` attributes are modified.

The rule is satisfied when there exists a dependence edge whose `extent` attribute is `nil`, there exists no < or > direction vectors associated with this edge, and the initial point of the dependence appears textually before the terminal point of the dependence.

The rule is satisfied at most once for each dependence edge since the `extent` attribute must be `nil`. The rule is satisfied only if all direction vectors are =. This corresponds to our previous definition of a loop-independent dependence.

Since a loop-independent dependence has no carrier loop, the `carrier` attribute is set to `0`.

For loop-carried dependencies, we must consider the set of plausible direction vectors. Recall from Definition 2.9 that the set of plausible direction vectors is, essentially, all direction vectors whose first direction vector that differs from = is <. Any dependence whose direction vectors are not plausible, is not a valid dependence.

The following rule fills in the `extent` and `carrier` attributes for valid dependencies.

Rule 2.5

```
(rule loop-carried
  {<ce> (dependence ^from <st1> ^to <st2> ^ref ARRAY_REF
     ^extent <extent> ^var <v> ^type <t> ^vfrom <vt1> ^vto <vt2>)}
  (direction ^dir '< ^dim <dim> ^var-from <vt1> ^var-to <vt2> ^depth <d>)
  (loop ^stmt <loop> ^depth <d>)
  -(direction ^dir << > < >> ^dim <dim> ^var-from <vt1> ^var-to <vt2>
     ^depth {< <d>})
  (statement ^stmt <st1> ^order <pos>)
```

```
   (statement ^stmt <st2> ^order {<= <pos>})
   -(dependence ^from <st1> ^to <st2> ^ref ARRAY_REF ^carrier <loop>
      ^extent loop-carried ^var <v> ^type <t> ^vfrom <vt1> ^vto <vt2>)
-->
   (if (eq?  <extent> 'nil)
    (modify <ce> ^extent loop-carried ^carrier <loop>)
    (make dependence ^from <st1> ^to <st2> ^ref ARRAY_REF ^type <t>
      ^var <v> ^vfrom <vt1> ^vto <vt2>
      ^extent loop-carried ^carrier <loop>)))
```

The negated condition element uses the "or" grouping operator, `<< >>`. Unfortunately, this makes the condition element more difficult to read. The element is matched only if memory does not contain any `>` or `<` direction vectors at a depth less than the first `<` direction. Condition elements two and three insure that the direction vector for the dependence is plausible.

The right hand side modifies the dependence edge so that the `extent` attribute is `loop-carried`. The `carrier` attribute is set to the statement tag for the loop at level of the leftmost `<` direction. If the dependence that triggers Rule 2.5 has already been processed, then a dependence edge is created for another dimension. This continues until an edge is created for each dimension.

Notice that this rule applies only to dependencies in subscripted variables. For scalar variables, the conditions for determining a loop-carried extent and the information required for finding the carrier loop is somewhat different. Information about loop nesting is also needed.

Loops are represented in $\mathcal{M}$ by the class `loop`. Loop representation is more complex than for other objects encountered so far. Since we need only two of the loop attributes here, will postpone the enumeration of all the attributes until Chapter 4.

Rule 2.6

```
(rule loop-carried-scalar
   {<ce> (dependence ^from <st1> ^to <st2> ^ref VAR_REF
      ^vfrom <vt1> ^vto <vt2> ^extent nil)}
   (statement ^stmt <st1> ^order <pos> ^depth <d1>)
```

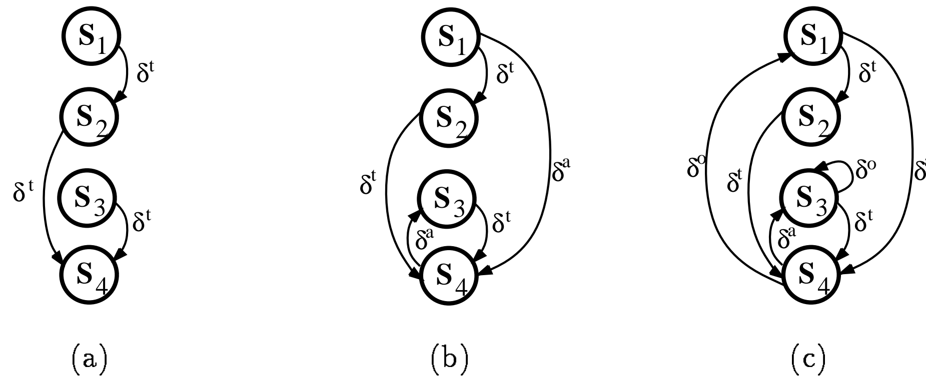

Figure 5: Derivation of the data dependence graph.

```
   (statement ^stmt <st2> ^order {<= <pos>} ^depth <d2>)
   (loop ^stmt <loop1> ^depth <d1>)
   (loop ^stmt <loop2> ^depth <d2>)
 -->
   (modify <ce> ^extent loop-carried
      ^carrier (if (<= <d1> <d2>) <loop1> <loop2>)))
```

Here the variables `<d1>` and `<d2>` are bound to the nesting depths of loops for the two statements involved in the dependence. The `carrier` attribute is set to the outermost of the two loops.

2.4.4 An Example

In this section we present a brief sketch of the DDG derivation for the following loop.

```
do i=1,100
 S1: a(i) = a(i+1)
 S2: b(2*i) = a(i)
 S3: c = b(2*i+1)
 S4: a(i+1) = b(2*i)+c
enddo
```

We assume that a parse tree has been constructed and the necessary information such as statement, variable, and instance tags, is readily available. Further, we assume that variable and flow analysis has been performed and the appropriate `var`, `flow`, `statement`, and `loop` elements are in $\mathcal{M}$. In what follows, we present the derivation of each type of dependence edge separately.

We begin with the true dependence edges (Figure 5a). The definition of `a(i)` in S_1 and the use of `a(i)` in S_2 along with the assertion from flow analysis that the definition in S_1 of `a` reaches the use of `a` in S_2 satisfies the preconditions of Rule 2.1. The dependence edge is added to $\mathcal{M}$. Subscript analysis enters a single `=` direction element into $\mathcal{M}$. Rule 2.4 is satisfied and the `extent` attribute is marked `loop-independent`.

The definition of `b(2*i)` in S_2 followed by the use of `b(2*i+1)` in S_3 results in a true dependence edge, but because the GCD Test proves that no solution exists for the dependence equation, no direction vectors are added to $\mathcal{M}$. Without direction vectors, no rules capable of filling in the `extent` can be satisfied and the dependence edge is never completed.

Finally, the definition of `c` in S_3 followed by the use of `c` in S_4 satisfies Rule 2.1. In addition to the presence of the dependence edge and the `loop` elements, Rule 2.6 is satisfied because S_4 textually follows S_3. Rule 2.4 modifies the `extent` attribute to `loop-independent` and sets the `carrier` attribute to the tag for the `do` statement.

The anti dependence edges (Figure 5b) are derived in a similar way. Rule 2.3 enters anti dependence edges from S_1 to S_4 on `a`, from S_2 to S_4 on `a`, from S_3 to S_2 on `b`, from S_4 to S_2 on `b`, and from S_4 to S_3 on `c`.

Rule 2.4 sets the `extent` attribute to `loop-independent` and the `carrier` attribute to `0` for the dependence from S_1 to S_4 on `a`.

For the dependencies from S_3 to S_2 on `b` and from S_4 to S_2 on `b`, Rule 2.5 sets the `extent` attribute to `loop-carried` and the `carrier` attribute to the tag value

for the loop.

Finally, for the dependence from S_4 to S_3 on `c`, Rule 2.6 sets the `extent` attribute to `loop-carried` and the `carrier` attribute to the tag value for the loop.

Notice that the dependence from S_2 to S_4 on `a` does not satisfy any of the rules. Since the direction vector associated with this dependence is `(>)`, this otherwise valid scalar dependence does not hold after subscript analysis.

The two output dependence edges (Figure 5c) are derived by the satisfaction of Rule 2.2 and Rule 2.5 for the loop-carried edge, and by Rules 2.2 and Rule 2.5 for the self-dependence edge.

2.5 Summary

In this chapter we discussed scalar and vector data dependence analysis and derived one well known dependence test, the GCD Test. We described a rule-based approach to deriving a data dependence graph and presented a set of rules that formally describe the meaning of data dependence. During our discussion, we illustrated the syntax and semantics of the experimental rule-based language, REX-UPSL.

Chapter 3

Recurrences

In this chapter we continue our look at the data dependence graph by developing rules to detect cycles. Further, we show how nodes can be grouped to form an acyclic dependence graph. This acyclic graph represents the sources of parallelism in the program. It is used in the following chapters to guide the restructuring process.

3.1 Introduction

When language level parallelism constructs are introduced into a program, either by vectorizing statements or by scheduling loop iterations for concurrent execution, cycles in the dependence graph must be considered.

The following loop contains a loop-carried dependence that induces a cycle in the dependence graph.

```
do i=1,64
 a(i) = a(i-1)*c
end
```

This loop is a tempting candidate for vectorization, but it cannot be vectorized. The semantics of vector machines require that all of the operands be available `prior` to the statement's execution. In this case, however, one operand depends on a previous iteration of the loop. This is an example of a broad class of *recurrence* computations in which there is a cycle in the dependence graph. Here the cycle is quite short, but

in general the cycle could include an arbitrary number of statements. Concurrent execution by vectorization of individual statements is inhibited by these cycles.

A cycle exists in the dependence graph when: 1) there is a path of dependencies from S_i to S_j, and 2) there is a loop-carried dependence from S_j to S_i. A loop-carried dependence is necessary because structured loops are the only iterative construct in our model.

Although the absence of loop-carried dependencies is clearly a sufficient condition allowing vectorization or parallelization, it is overly conservative. Using this condition alone may cause a restructurer to abandon an otherwise useful restructuring plan. In this chapter, we present results that allow a rule-based restructurer to use both necessary conditions to increase the efficacy of the restructuring process.

In the next section we formalize the notion of "a dependence path from S_i to S_j" and present two rules for finding these paths. In Section 3, we discuss a method that a rule-based restructurer can use to detect and represent recurrences. The restructuring rules in the next chapter will use information about these recurrences to guide the restructuring process. They will also dynamically update the recurrence representations after recurrence elimination operators have been applied.

Inevitably, some recurrences are likely to remain in the program after the restructuring process. Because there may be dependencies between statements in different recurrences there is, ultimately, an order in which the recurrences themselves must be executed. In Section 4, we present a set of rules to partition the dependence graph into π-*blocks*. This forms an acyclic graph that can be topologically sorted to produce a semantically valid statement ordering.

In the final section we present a single, powerful rule that performs the topological sort and produces a semantically valid statement ordering.

3.2 Transitive Closure of δ

In Section 2.2 we introduced the notion of a dependence path between two statements. This illustrated the idea of the transitive closure of the δ relation. In this section, we examine this issue in depth and present a two rules to derive the transitive closure of the δ relations represented in a dependence graph. In the next section we use this new relation to detect the recurrences in the dependence graph.

We begin with the definition.

Definition 3.1 *Define the δ^+ relation inductively as,*

$$S_i \ \delta^+ \ S_j \quad \textit{iff} \quad \begin{cases} S_i \ \delta \ S_j \ \ \textit{or}, \\ S_i \ \delta^+ \ S_n \ \wedge \ S_n \ \delta \ S_j \quad \textit{where } i \neq n \neq j \end{cases}$$

The δ^+ relation represents a chain of data dependencies between statements S_i and S_j. Notice that Definition 3.1 uses the generic dependence relation δ, hence the definition of δ^+ does not depend on the types of data dependencies along the path from S_i to S_j.

We represent the δ^+ relation in $\mathcal{M}$ with the class `dependence+`. The attributes for `dependence+` are:

- `path` is a list of statement tags along path.
- `from` is the statement tag of initial point in the path.
- `to` is the statement tag of terminal point in the path.

Once all the edges of the dependence graph have been derived, the following rules are employed to derived the δ^+ relations in $\mathcal{M}$. We have a separate rule for each part of Definition 3.1.

Rule 3.1

```
(rule base-case
  (dependence ^from <st1> ^to {<st2> <> <st1>} ^extent <> nil)
  -(dependence+ ^from <st1> ^to <st2>)
-->
  (make dependence+ ^from <st1> ^to <st2> ^path (list <st1> <st2>)))
```

This rule handles the base case of Definition 3.1. The second condition element guarantees that the rule is satisfied exactly once for each pair S_i, S_j where $S_i \; \delta \; S_j$.

Rule 3.2

```
(rule induction-step
  (dependence+ ^from <st1> ^to <st2> ^path <others>)
  (dependence ^from <st2> ^to {<st3> <> <st2> <> <st1>} ^extent <> nil)
-->
  (unless (in?  <st3> <others>)
   (make dependence+ ^from <st1> ^to <st3> ^path (cons <st3> <others>))))
```

This rule handles the induction step of Definition 3.1. The second condition element uses the `<>` predicate and the grouping brackets to insure that the $i \neq n \neq j$ condition of Definition 3.1 is satisfied. On the right hand side, the `unless` special form guarantees that the terminal point of the proposed new for δ^+ relation is not already a member of the path. This prevents cycles in the closure derivation.

3.3 Finding Recurrences

Cycles in the dependence graph arise from recurrence computations in the program. To simplify matters, we will use the term recurrence to mean a cycle in the dependence graph.

Definition 3.2 *If $\exists \, i, j : S_i \; \delta^+ S_j \; \wedge \; S_j \; \delta \; S_i$, then we say that there is a recurrence in the dependence graph.*

The following theorem states that a loop-carried dependence is a required for a recurrence in the dependence graph.

Theorem 3.1 *A recurrence exists in a dependence graph if and only if*

$$\exists\, i, j : S_i\ \delta^+ S_j\ \wedge\ S_j \delta_c\ S_i$$

Proof If there exists i, j such that $S_i\ \delta^+ S_j\ \wedge\ S_j \delta_c\ S_i$, then by Definition 3.2, there exists a recurrence in the graph.

If there is a recurrence in the dependence graph, then by Definition 3.2 there exists statements S_i and S_j such that $S_i\ \delta^+ S_j\ \wedge\ S_j \delta\ S_i$. Without lost of generality, we may assume that there are no loop-carried dependencies in along the dependence path $S_i\ \delta^+ S_j$. If there are, then we are done. If there are not, then by Definition 2.6 and Definition 3.1, the execution of S_i precedes the execution of S_j since $S_i\ \delta^+\ S_j$. By Definition 2.6 and $S_j\ \delta\ S_i$, the execution of S_j precedes the execution of S_i. Since the only iterative construct in our model is the structured loop, the dependence from S_j to S_i is loop-carried. □

3.3.1 Representing Recurrences

Representing recurrences in $\mathcal{M}$ is somewhat more involved because each recurrence can have an arbitrary number of members. If $\mathcal{M}$ were a strictly relational database, each $\mathcal{M}_e$ could contain only atomic values and $\mathcal{M}$ would then be a collection of normalized relations [27]. Although normalized relations lead to simpler rules, using lists in $\mathcal{M}_e$ has advantages for the imperative code. For our representation of recurrences we use a normalized form for the benefit of the rules, and an unnormalized form for the benefit of the right hand side expressions.

We represent recurrences in $\mathcal{M}$ by the element class `recurrence`. Each member of a recurrence is represented by the class `recurrence-member`. The attributes associated with the class `recurrence` are:

- `id` is a unique recurrence identifier.

- `from` is a statement tag of the initial point of the loop-carried dependence.
- `to` is statement tag of the terminal point of the loop-carried dependence.
- `members` is a list of members of the recurrence.
- `vname` is the variable involved in the loop-carried dependence.

The attributes for the class `recurrence-member` are:

- `id` is a unique recurrence identifier.
- `stmt` is a statement tag of the member.
- `carrier` is a statement tag of the carrier loop.

The intent here is that each recurrence has a unique symbol associated with its `id` attribute. The same symbol is associated with each of the `recurrence-member`'s `id` attributes. In relational database terms, this symbol is the *foreign key* by which access is gained to individual members of the recurrence.

The `members` attribute, however, violates the normal form requirement. This makes $\mathcal{M}$ an invalid relational database, although it is still a collection of mathematical relations.

As an example of recurrence representation in $\mathcal{M}$, consider the following set of relations: $S_1 \; \delta \; S_2, S_2 \; \delta \; S_3$, and $S_3 \; \delta_1 \; S_1$. Here we have a recurrence involving three statements S_1, S_2, and S_3, with a loop-carried dependence at depth 1 between S_3 and S_1. This recurrence is represented as:

```
(recurrence ^id U76 ^from t_S3 ^to t_S1 ^members (t_S1 t_S2 t_S3) ^vname t_v)
(recurrence-member ^id U76 ^stmt t_S1 ^depth 1)
(recurrence-member ^id U76 ^stmt t_S2 ^depth 1)
(recurrence-member ^id U76 ^stmt t_S3 ^depth 1)
```

The recurrence identifier, `U76`, is the key with which each of the recurrence members is accessed. We will use a similar representational scheme for π-blocks.

3.3.2 Deriving Recurrences

From Theorem 3.1 we know that the necessary conditions for a recurrence is a chain of dependencies between two statements and loop-carried dependence from the terminal point to the initial point in the chain. The simplest case that satisfies these conditions is a loop-carried self-dependence. The following rule handles this case.

Rule 3.3

```
(rule build-single-recurrences
   (dependence ^var <v> ^extent loop-carried ^from <st> ^to <st>
      ^carrier <loop> ^vfrom <vt1> ^vto <vt2>)
-->
   (let ([<id> (gensym)])
    (make recurrence ^id <id> ^members (cons <st> nil)
      ^from <st> ^to <st> ^vname <v>)
   (make recurrence-member ^id <id> ^stmt <st> ^carrier <loop>)))
```

The variables associated with the `from` and `to` attributes are the same. This forces these variables to be bound to the same value. The `extent` attribute of the dependence is `loop-carried`.

The right hand side invokes the `gensym` special form to create a unique symbol to be used as the recurrence identifier. This identifier is associated with the `id` attribute of both the `recurrence` element and the `recurrence-member` element.

For dependence chains longer than one element and an accompanying loop-carried dependence, the following rule constructs the representation of the recurrence in $\mathcal{M}$.

Rule 3.4

```
(rule build-multi-recurrences
   (dependence ^var <v> ^extent loop-carried ^from <st1> ^to <st2>
      ^carrier <loop> ^vfrom <vt1> ^vto <vt2>)
   (dependence+ ^from <st2> ^to <st1> ^path <others>)
-->
```

```
(let ([<id> (gensym)])
 (make recurrence ^id <id> ^members <others> ^from <st1>
     ^to <st2> ^vname <v>)
 ((rec loop
   (lambda (l)
     (if (null?  l) nil
       (begin
         (make recurrence-member ^id <id> ^stmt (car l) ^carrier <loop>)
         (loop (cdr l)))))) <others>)))
```

The right hand side generates and binds a new symbol to the variable `<id>`. This symbol is associated with the `id` attributes of the `recurrence` element and each of the `recurrence-member` elements.

3.4 π-Blocks: Building an Acyclic DDG

It is important to node that every recurrence is found by the previous two rules. Some recurrences might be subcycles of other recurrences. While some recurrences are guaranteed to be maximal recurrences. The set of maximal recurrences are called π-blocks [46, 49].

Given the set of π-blocks for a dependence graph G, we can construct a new graph G' whose nodes are the π-blocks of G and whose edges are the dependencies between statements in different π-blocks. G' represents the strongly connected components of G. The directed edges in G' impose a partial ordering on the execution of π-blocks. G' is clearly an acyclic graph. A topological sort yields a valid execution ordering for the π-blocks. We will see in the Chapter 6 that a series of transformation operators may result in a different set of π-blocks and a different execution ordering.

We use a representation for π-blocks that is similar to recurrences. The class `pi-block` represents the π-blocks in $\mathcal{M}$ while the class `pi-block-member` represents the members of the π-block. The attributes for the class `pi-block` are:

- `id` is a unique π-block identifier.

- `seq` is a sequence number used in ordering π-blocks.

The attributes for class `pi-block-member` are:

- `id` is a unique π-block identifier.
- `stmt` is a statement tag of the member.
- `cnt` is a recurrence reference count.

To build the set of π-blocks, we need to find the strongly connected components of the dependence graph. In an imperative environment, Tarjan's algorithm [63] would be best. For our purposes, a more computationally expensive, yet simpler approach will do. The following lemma serves as the basis for our approach.

Lemma 3.1 *If two recurrences have a statement in common, then the union of statements in the recurrences form a larger recurrence.*

Proof Given two recurrences, A and B and a statement S in common, there is a path from a node in A to node S in B. Conversely, there is a path from a node in B to a node S in A. By Definition 3.2 there is a path from any statement in A to any statement in B and from any statement in B to any statement in A. Hence, $A \cup B$ is a recurrence. □

Essentially, our approach is to merge recurrences until we have disjoint sets. Although this is not the most computationally efficient approach, it is suitable for the rule-based programming paradigm.

We use two new element classes, `merge` and `merged`. The class `merge` has the attributes:

- `into` is the π-block identifier to merge into.
- `from` is a recurrence identifier to merge from.

The class `merged` as the attributes:

- `id` is a recurrence identifier.
- `stmt` is the tag of merged statement.

We begin by creating a `pi-block` for each statement that is not a member of a recurrence.

Rule 3.5
```
(rule make-statements
  (statement ^stmt <st> ^class {<> COMMENT <> IF <> LOGIF})
  -(recurrence-member ^stmt <st>)
-->
  (let ([<id> (gensym)])
   (make pi-block ^id <id> ^seq (set!  *sequence* (+ 1 *sequence*)))
   (make pi-block-member ^id <id> ^stmt <st> ^cnt 1)))
```

Next, we generate subgoals to merge those statements that are members of exactly one recurrence into an (as yet) unrepresented π-block.

Rule 3.6
```
(rule no-merge
  (recurrence-member ^id <id> ^stmt <st>)
  -(recurrence-member ^id {<> <id>} ^stmt <st>)
  -(merge ^from <id>)
-->
  (make merge ^from <id> ^into (gensym)))
```

For recurrences that are not disjoint, the following two rules generate a subgoals to merge the recurrences into common π-blocks.

Rule 3.7
```
(rule merge-starter
  (recurrence-member ^id <id1> ^stmt <st>)
  (recurrence-member ^id {<id2> <> <id1>} ^stmt <st>)
  -(merge ^from <id1>)
  -(merge ^from <id2>)
-->
  (let ([<pbid> (gensym)])
   (make merge ^from <id1> ^into <pbid>)
   (make merge ^from <id2> ^into <pbid>)))
```

Rule 3.8

```
(rule merge-odd
  (recurrence-member ^id <id> ^stmt <st>)
  (pi-block-member ^id <pbid> ^stmt <st>)
  -(merge ^from <id> ^into <pbid>)
-->
  (make merge ^from <id> ^into <pbid>))
```

Finally, for each `merge` subgoal, the following three rules carry out the indicated merge, creating `pi-blocks` where necessary.

Rule 3.9

```
(rule merge-first
  (merge ^from <id> ^into <pbid>)
  (recurrence-member ^id <id> ^stmt <st>)
  -(pi-block-member ^id <pbid> ^stmt <st>)
-->
  (make merged ^id <id> ^stmt <st>)
  (make pi-block-member ^id <pbid> ^stmt <st> ^cnt 1))
```

Rule 3.10

```
(rule merge
  (merge ^from <id> ^into <pbid>)
  (recurrence-member ^id <id> ^stmt <st>)
  {<ce> (pi-block-member ^id <pbid> ^stmt <st> ^cnt <c>)}
  -(merged ^id <id> ^stmt <st>)
-->
  (make merged ^id <id> ^stmt <st>)
  (modify <ce> ^cnt (+ 1 <c>)))
```

Rule 3.11

```
(rule make-pi-block
  (merge ^from <id> ^into <pbid>)
  -(pi-block ^id <pbid>)
-->
  (make pi-block ^id <pbid> ^seq (set!  *sequence* (+ 1 *sequence*))))
```

3.5 π-Block Order

In the previous section we alluded to ordering π-blocks of a dependence graph to obtain valid execution order. Since the π-blocks together with the dependence edges

form an acyclic graph, a topological sort is possible. Given the `order-statements` subgoal, the following rule is satisfied until no two π-blocks are out of order relative to their `seq` attributes.

Rule 3.12

```
(rule sort
  {<ce0> (order-statements)}
  {<ce1> (pi-block ^id <id1> ^seq <pos1>)}
  {<ce2> (pi-block ^id {<id2> <> <id1>} ^seq {<pos2> < <pos1>})}
  (pi-block-member ^stmt <st1> ^id <id1>)
  (pi-block-member ^stmt <st2> ^id <id2>)
  (dependence ^from <st1> ^to <st2>)
-->
  (remove <ce0>)
  (make order-statements)
  (modify <ce1> ^seq <pos2>)
  (modify <ce2> ^seq <pos1>))
```

The second, third, and fifth condition elements on the left hand side are critical. These are satisfied if and only if there exists two distinct (note the `<id2> <> <id1>` clause) π-blocks in an ordering that contradicts a `dependence` assertion. The `modify` actions on the right hand side change the ordering change the ordering to conform to the dependence assertion.

For reasons discussed in Chapter 6, the first condition element on the left hand side together with the first expression on the right hand side serve to keep the inference engine focused on this rule. Without this focus, other rules, possibly some that add or delete π-blocks, could be selected for execution while this rule is active.

3.6 Summary

In this chapter we have presented a representational scheme for dependence paths, recurrences, and π-blocks. We have developed a series of rules for deriving the dependence paths, detecting recurrences, and partitioning the graph into π-blocks.

In the next chapter we use the derived dependencies, recurrences, and π-blocks to guide the transformation operator selection process.

Chapter 4

Transformation Operators

The previous chapters provided us with the theory necessary to insure semantic invariance of a program after the application of one or more transformation operators. They also provided us with a representation schema for, and the rules to derive, data dependencies, recurrences, and π-blocks.

This chapter begins our discussion of the transformation operators. Here we present a detailed look at several transformation operators. Our discussion focuses on the preconditions that must exist before an operator can be applied. We also pay special attention to the conditions that suggest a particular operator. These conditions play an important role in selecting operators and planning a sequence of operators.

4.1 Introduction

The restructuring process consists of selecting and applying a series of transformation operators. The goal of the restructuring process is to improve the match between the programming language and the underlying hardware. The process is, in part, one of finding and exploiting parallelism already present in the original serial program. But, finding sources of parallelism is not the whole story. The match can also be improved, and thus the performance improved, by increasing data localization on non-uniform memory access machines [51], optimizing cache performance [36], re-using registers,

and other optimizations.

In a series of transformation operators, each operator either improves the program-machine match or it converts the program into a form suitable for the application of another operator. For each operator there is a set of indications that suggest the operator's use. The use of a particular operator is indicated by either its direct improvement of the match or its ability to convert the program into a form required by some other operator.

Also associated with some operators is a set of counter-indications. These are conditions, such as a recurrence or the absence of a particular machine property, that prevent the application of a given operator.

Indications can be viewed as a first approximation to the usefulness requirement, while counter-indications are a similar approximation of the validity requirement.

In this chapter we examine in detail a small, but important, set of transformation operators. For each operator there is a set of preconditions that must be satisfied by $\mathcal{M}$ for the operator to be applied. Some preconditions are complex enough to require their own rules, others can be specified in the condition elements on the left hand side of the rule that selects the transformation operator. In either case, encapsulating preconditions in rules serves not only to modularize the preconditions, but also to formalize them.

In the next chapter, we use the transformation operators discussed here to illustrate a hierarchical, rule-based planning strategy.

4.2 Properties

The restructuring process attempts to convert one program representation into another, semantically equivalent, representation that more closely matches the properties of the target machine. This means that we must account for the target machine's

Class	Property
`vector-capable`	Has vector hardware
`parallel-capable`	Has parallel hardware
`allow-vector-masks`	Has vector masking
`allow-nonunit-stride`	Strides other than 1 allowed
`vector-length`	Specifies min, max, and best lengths
`grain-size`	Specifies min and best grain sizes

Table 2: Machine property classes in $\mathcal{M}$.

properties. Implicitly, it also means we must account for the program's properties. In this section we discuss both of these issues and develop a representation for properties in $\mathcal{M}$.

4.2.1 Machine Properties

For each target machine there is a vast array of properties that describe architectural features of the machine. Some of these properties are specific to the machine. Some are specific to the class of the machine (SIMD, MIMD, etc).

Whatever the source of the properties, some are essential in the restructuring process. Whether the machine is capable of vector or parallel processing, for example, is clearly essential information. The minimum, maximum, and optimum vector sizes, the feasibility of non-unit strides, and the vector masking capabilities are important for vector processors. For parallel processors, the minimum and optimal grain sizes are important. These are just a few of the many useful machine properties, others can be found in the literature [68, 43, 28, 18, 65, 22, 37, 30]. In order to simplify the discussion, we consider only a small set of properties (see Table 2).

Some of these properties, like the best vector length, are only rough first approximations. Properties such as this often depend on more than what can be derived from a static, analytic model of the target machine. Indeed a useful avenue of research is

to develop rule-based reasoning methods that combine information obtained from an analytic machine model and indepth semantic analysis of the program to derive more accurate sets of machine properties.

4.2.2 Program Properties

Results in the previous two chapters provide us with almost all the program properties we need. We presented both representations and derivation rules for data dependencies, recurrences, and π-blocks. Other properties, such as statement classifications and control dependencies[1] were derived from a traditional parse of the program.

Representing Loop Properties

Loops can be nested to arbitrary depths. Loops have lower and upper bounds along with some increment value. Associated with a loop, is some collection of statements, often called the loop body.

As with recurrences and π-blocks, loops represent a non-flat relational domain. To simplify its use in rules, we normalize the representation using a unique symbol as a foreign key to access members of the loop.

A loop is represented in $\mathcal{M}$ by the class `loop`. The attributes for `loop` are:

- `id` is the unique loop identifier.
- `stmt` is the statement tag of the loop header.
- `depth` is the nesting level of this loop.
- `lower` is the lower bound for the loop.
- `upper` is the upper bound for the loop.

[1] Recall that the control parent of each statement is contained in the `statement` class.

- `step` is the increment value for the loop.
- `request-id` is used in updating loop information.

Each statement in the body of the loop is represented by the class `loop-member`. The attributes for `loop-member` are:

- `id` is the unique loop identifier.
- `stmt` is the tag for this statement.
- `parallelizable` is `yes` if the statement is parallelizable.

Finally, for each loop we maintain the separate class `loop-profile`.

- `id` is the unique loop identifier.
- `stride` is `ok` if the stride is acceptable.
- `length` is `ok` if the loop length is acceptable.

Deriving Loop Properties

The `loop-profile` class contains the derived properties `stride` and `length`. The following rules use machine properties along with the loop properties to derive these values.

Rule 4.1

```
(rule nonunit-stride-check
  (loop ^id <id> ^step <> 1)
  {<ce> (loop-profile ^id <id> ^stride nil)}
  (allow-nonunit-stride)
-->
  (modify <ce> ^stride ok))
```

In this rule, the `stride` attribute is marked `ok` if the loop is a non-unit stride, but such strides are allowed by the hardware. For some machines, such as the Cyber 205, the use of non-unit strides require expensive scatter/gather operations [43]. Often, the cost of these operations may dominate the performance enhancement gained by vectorization. For machines like the Cyber, this rule might be modified to allow non-unit strides if the length of the vector operation is above the some break even threshold.

Rule 4.2

```
(rule unit-stride-check
  (loop ^id <id> ^step 1)
  {<ce> (loop-profile ^id <id> ^stride nil)}
-->
  (modify <ce> ^stride ok))
```

If the stride is 1, then this rule modifies the `stride` attribute of the loop's profile.

Rule 4.3

```
(rule length-check-constant
  (loop ^id <id> ^length <c>)
  (vector-length ^min {<= <c>})
  {<ce> (loop-profile ^id <id> ^length nil)}
-->
  (modify <ce> ^length ok))
```

Rule 4.4

```
(rule length-check-expr
  (loop ^id <id> ^length EXPR)
  {<ce> (loop-profile ^id <id> ^length nil)}
-->
  (modify <ce> ^length ok))
```

These rules modify the `length` attribute if the loop length (trip count) is greater than or equal to the minimum allowable vector length, or the loop length is not known at compile time. The `length` attribute is used to determine if vectorization will improve performance.

The startup costs for a vector instruction is always greater than the cost to produce a single result once the pipeline is full. These startup costs are often quite high. If the number of instructions to be executed by a single vector instruction is below some minimum threshold, there will be a net performance loss in using the vector instruction as opposed to a traditional loop. This threshold is highly machine dependent. It also can depend on the vector instruction, state of the cache, and many other factors. These two rules represent only a first approximation to the general problem of finding the break even point.

Program References

Two additional program properties are useful. The class `function-reference` lists each statement that contains a reference to function. The attributes are:

- `function` is the function's name.
- `stmt` is the statement's tag.

The class `reference` distinguishes each variable reference within each statement as either a scalar reference, denoted by `VAR_REF`, or array reference, denoted by `ARRAY_REF`. The attributes are:

- `var` is the referenced variable's name
- `stmt` is the statement's tag.
- `var-type` is either `ARRAY_REF` or `VAR_REF`

4.3 Vectorization

In terms of potential performance enhancement, vectorization is one of two primary transformation operations that improve the match between a serial program and a supercomputer class target machine. The other primary transformation, parallelization, is discussed in the next section. In this section, we discuss a rule-based scheme for finding potential vectorization candidates and applying the vectorization transformation operator. We discuss methods for classifying statements as vectorizable, selecting statements that meet the preconditions for vectorization, and selecting operators to create the preconditions for vectorization.

4.3.1 Finding Vectorizable Statements

Not all statements are candidates for vectorization. Clearly, statements that perform I/O, invoke functions, or have arbitrary subscript expressions, are not vectorizable. These restrictions seem rather severe. One might think that few statements are vectorizable. But, in practice, many statements can be vectorized.

The following definition makes clear precisely what we mean by a vectorizable statement.

Definition 4.1 *A statement $S \in P$ in program* P *is vectorizable if and only if all of the following are true,*

- *The machine is vector capable.*
- *S is an assignment statement.*
- *S is a statement inside a loop.*
- *S contains at least one array reference whose subscript expression contains the induction variable for the loop.*

- *The stride for the innermost loop containing S is allowed.*
- *The length (trip count) of the innermost loop containing S is above the threshold limit.*
- *S contains no function references.*
- *The vectorization of S is not explicitly inhibited.*[2]

Definition 4.1 leads directly to the following rule.

Rule 4.5

```
(rule vectorizable
  (vector-capable)
  (vectorize-phase)
  (statement ^class ASSIGN ^stmt <st> ^depth <d>)
  (reference ^stmt <st> ^var-type ARRAY_REF)
  (loop ^id <id> ^depth <d>)
  (loop-profile ^id <id> ^length ok ^stride ok)
  -(function-reference ^stmt <st>)
  -(vectorizable ^stmt <st>)
  -(inhibit-vectorization ^stmt <st>)
-->
  (make vectorizable ^stmt <st>))
```

This rule adds an element of the class `vectorizable` for each statement in region R of program P that is vectorizable by Definition 4.1. In the next section we will see that a precondition for the `vectorize` transformation operator is a corresponding `vectorizable` element in $\mathcal{M}$.

4.3.2 Vectorizing Operators

In the restructuring phase in which the rules of this chapter are active, the transformation operators are represented in $\mathcal{M}$. For reasons discussed in the next chapter,

[2]Inhibiting vectorization, discussed in the next chapter, is used during backtracking in planning.

the operators are not applied immediatedly. In this section we discuss the derivation and representation of two vectorizing operators.

The `vectorize` Operator

As mentioned previously, a statement may not be vectorized if it is a member of a recurrence carried by the innermost loop. For vectorizable statements outside of `IF` statements that are not members inner-loop recurrences, the following rule adds the element class `vectorize` to $\mathcal{M}$.

Rule 4.6

```
(rule vectorize
  (vectorizable ^stmt <st>)
  (statement ^stmt <st> ^control <cpid> ^depth <d>)
  (loop ^stmt <loop> ^depth <d>)
  -(recurrence-member ^stmt <st> ^carrier <loop>)
  -(statement ^stmt <cpid> ^class << IF LOGIF >>)
  -(vectorize ^stmt <st>)
-->
  (make vectorize ^stmt <st>))
```

The `vectorize-with-guard` Operator

`IF` statements denote conditional execution of one or more statements. A direct vectorization of a statement in the body of an IF may change the result of the loop because during some iterations, the statement may not be executed. Some machines are capable of conditional vector execution. This property is represented by the element class `allow-vector-masks`. The following rule is identical to Rule 4.6, except that it *requires* the control parent to be an IF statement and the machine property `allow-vector-masks`. The rule generates a `vectorize-with-guard` element.

Rule 4.7

```
(rule vectorizable-guard
   (vectorizable ^stmt <st>)
   (allow-vector-masks)
   (statement ^stmt <st> ^control <cpid> ^depth <d>)
   (loop ^stmt <loop> ^depth <d>)
   -(recurrence-member ^stmt <st> ^carrier <loop>)
   (statement ^stmt <cpid> ^class << IF LOGIF >>)
   -(vectorize-with-guard ^stmt <st>)
 -->
   (make vectorize-with-guard ^stmt <st> ^guard-stmt <cpid>
      ^guard-var (generate-temporary-variable)))
```

The intent here is that a loop of the form,

```
do i=1,100
   if (k(i).gt.10) a(i) = a(i)*k(i)
enddo
```

is vectorized as,

```
do i=1,100
   %k(i) = k(i).gt.10
enddo
where(%k(1:100)) a(1:100) = a(1:100)*k(1:100)
```

The variable `%k` is a compiler generated mask array that holds the value `.true.` for each element of `k` that is greater than 10. The `where` clause is a guard that allows the element by element assignment to array `a` to succeed only when the corresponding element of the mask array is `.true.`.[3]

This generates an additional statement which in turn can sometimes be vectorized. But it also generates a dependence from the mask array to the new vector statement.

[3] The `where` clause is part of the FORTRAN 90 standard.

This technique of converting the control dependence to a data dependence is related to if-conversion [5] used in PFC [8].

The following rule adds dependence created by the if-conversion.

Rule 4.8

```
(rule if-conversion
  (vectorize-with-guard ^stmt <st> ^guard-var <guard> ^guard-stmt <if>)
  -(vectorize-if ^stmt <if>)
-->
  (make dependence ^from <if> ^to <st> ^extent loop-independent
     ^var <guard> ^type true ^ref ARRAY_REF)
  (make vectorize-if ^stmt <if> ^guard-var <guard>))
```

4.3.3 Changing Recurrences

Both Rule 4.6 and Rule 4.7 require that the statement not be part of recurrence carried by the innermost loop. In this section, we examine two transformation operators, loop interchange and scalar expansion. Loop interchange can be used to move a recurrence to an outer loop, while scalar expansion can be used to break a recurrence. In either case, the change can allow some statements to be vectorized.

Loop Interchange

Wolfe [72], Allen and Kennedy [6], and Banerjee [11] have studied the loop interchange problem extensively. Loop interchange is widely used in restructuring compilers primarily because it represents a powerful method with which one can exploit statement level parallelism, reduce memory bank conflicts, or increase register utilization. A loop interchange transformation takes a loop of the form,

```
L1: do i=1,3
L2:  do j=1,3
      a(i,j) = a(i+1,j+1)
     enddo
    enddo
```

and converts it into a loop of the form,

```
L2: do j=1,3
L1:  do i=1,3
      a(i,j) = a(i+1,j+1)
     enddo
    enddo
```

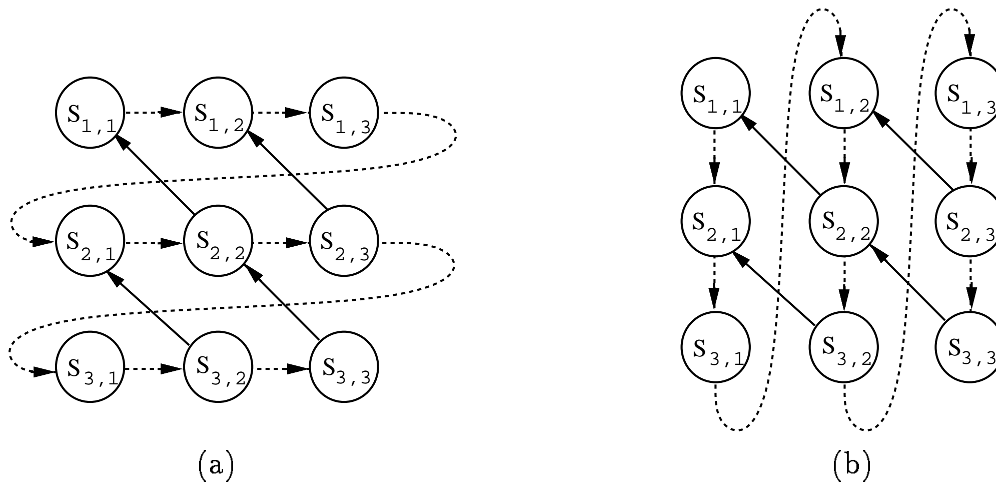

Figure 6: Dependencies before and after valid loop interchange.

Loop interchange is perhaps the most basic of all restructuring transformations. It is implemented in nearly every commercial restructuring compiler. It is most profitable in vectorization when an inner loop carries a dependence that can be exchanged

for an outer loop that does not carry a dependence. The inner loop can then be vectorized if the affected loop-carried dependence was part of a recurrence that had previously inhibited vectorization.

In some cases, a loop permutation changes the grain size of a parallelized outer loop. By increasing the grain size, the startup and scheduling overhead can be justified by larger units of work.

Interchanging loops also interchanges the loop bounds on the innermost loop. This can change a loop's stride through memory, often an important consideration for optimizing memory access.

Together with strip mining, loop permutation can be used to optimize cache usage by increasing the locality of data reference [36, 1].

Interchanging loops may increase the size of the vector, hence reduce the startup overhead. For example,

```
do i=1,100
 do j=1,5
  a(i,j) = a(i,j)*b(i)+c(i,j)
 enddo
enddo
```

can be vectorized, but the resulting code executes 100 small vectors. Interchanging the loops and vectorizing yields code the executes 5 long vectors instructions. The impact of the startup overhead is significantly lower.

It is important to note that no new statements are executed and no statements that are executed in the original loop are left out of the restructured loop. All the dependencies that exist in the original loop remain in the restructured loop. However, the *direction* of some of the dependencies may change. If a loop interchange results in changing the direction of some or all of the dependencies, the transformation is invalid.

Consider again the previous example of loop interchange.

```
L1: do i=1,3
L2:  do j=1,3
      a(i,j) = a(i+1,j+1)
     enddo
    enddo
```

Figure 6 makes this notion of direction change more clear. Figure 6a shows the original dependencies. The statement instances are represented by nodes while the dependencies are represented by solid directed edges. Dotted directed edges represent the execution flow between statement instances.

```
L2: do j=1,3
L1:  do i=1,3
      a(i,j) = a(i+1,j+1)
     enddo
    enddo
```

After L_1 and L_2 are interchanged (Figure 6b), each instance retains the same dependence relation, hence the loop interchange is valid.

Consider the following, slightly different loop,

```
L1: do i=1,3
L2:  do j=1,3
      a(i,j) = a(i-1,j+1)
     enddo
    enddo
```

and the corresponding loop interchange,

```
L2: do j=1,3
L1:  do i=1,3
      a(i,j) = a(i-1,j+1)
```

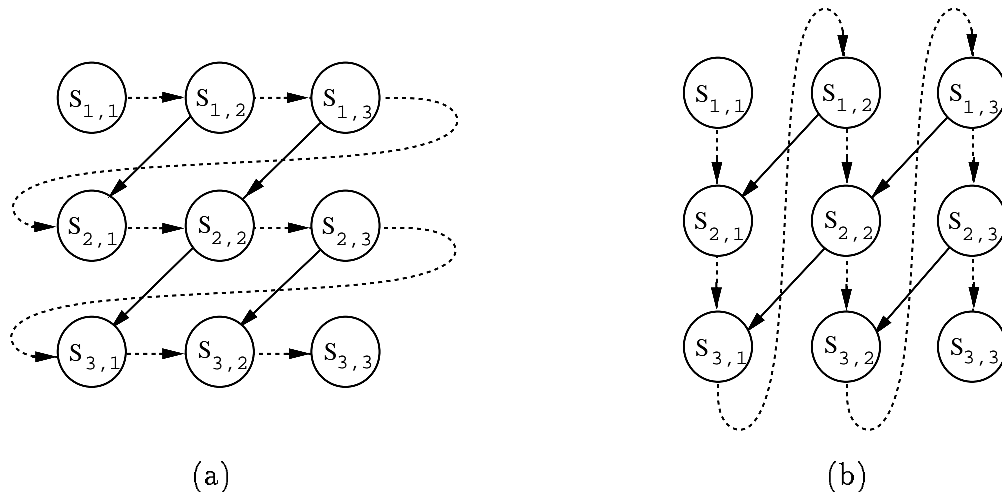

Figure 7: Dependencies before and after invalid loop interchange.

```
  enddo
enddo
```

The dependencies are illustrated in Figures 7a and 7b. Notice that the loop interchange changes a true dependence into an anti dependence. This transformation is invalid.

What preconditions must exist in order for a particular loop interchange to be valid? This question is examined in detail in the literature. The answer can essentially be formulated in terms of direction vectors as follows. Each loop interchange results in a corresponding permutation of the direction vectors of each of the dependence relations associated with the statements in the body of the loops. If the permutation changes the direction of a dependence, then the interchange is invalid.

Lemma 4.1 *If a loop interchange changes leftmost loop-carried direction of any direction vector from a* $<$ *direction to a* $>$ *direction, then the loop interchange is invalid.*

Proof See [11]. □

For example, a pair of loops with direction vectors, {(=,<),(<,>)}, could not be interchanged because the direction vector would change from (<,>) to (>,<). Here the leftmost loop-carried direction is changed from < to >, hence the interchange is invalid.

Although we have mentioned several reasons for loop interchange, the emphasis of this section is changing recurrences to enable vectorization. Here the primary reason for loop interchange is to move a recurrence to an outer loop, thus allowing vectorization of an inner loop. The following rule is satisfied by these conditions as well as the conditions of Lemma 4.1.

Rule 4.9

```
(rule loop-interchange
  (vectorizable ^stmt <st>)
  (statement ^stmt <st> ^depth <d1>)
  {<ce1> (loop ^stmt <inner> ^depth <d1>)}
  (recurrence-member ^stmt <st> ^id <id> ^carrier <inner>)
  (recurrence ^id <id> ^from <st1> ^to <st2> ^vname <v>)
  (dependence ^var <v> ^ref ARRAY_REF ^from <st1> ^to <st2>
     ^vfrom <vt1> ^vto <vt2> ^carrier <inner>)
  (direction ^dir '< ^depth <d1> ^dim <dim> ^var-from <vt1> ^var-to <vt2>)
  (direction ^dir '= ^depth <d2> ^dim <dim> ^var-from <vt1> ^var-to <vt2>)
  {<ce2> (loop ^stmt <outer> ^depth <d2>)}
  -(interchange-loops ^this <d1> ^that <d2>)
  -(direction ^dir '> ^depth <d1> ^carrier-depth >= <d2>)
  -(direction ^dir <> '= ^depth <d2> ^dim <> <dim>
     ^var-from <vt1> ^var-to <vt2>)
-->
  (modify <ce1> ^depth <d2>)
  (modify <ce2> ^depth <d1>)
  (make interchange-loops ^this <d1> ^that <d2> ^request-id (gensym)
     ^reason vectorize ^stmt <st> ^id <id>
     ^inner <outer> ^outer <inner>))
```

The left hand side is long, but not overly complex. The first six condition elements are satisfied if a vectorizable statement is a member of a recurrence carried by the innermost loop.

The next two condition elements are satisfied when: 1) there exists a < direction associated with the dependence carrying loop, and 2) there is a = direction in the same dimension associated with an outer loop. The reason that we require a = direction in some dimension associated with an outer loop is that any other direction would maintain the recurrence at its current level and the loop-interchange would not be useful.

The first of the final two condition elements requires that there exist no > direction whose carrier loop is between the outer and inner loop, inclusive. This guarantees that the interchange does not move a > direction to the left of any < direction.[4]

The last condition element prevents a pathological case. Consider the following loop with the direction vector, (=,<) for one dimension and the direction vector, (<,=) in the other dimension.

```
L1: do j=2,n
L2:  do i=2,n
      a(i,j) = a(i-1,j-1)+b(i)
     enddo
    enddo
```

Without the last condition element, Rule 4.9 would interchange L_1 and L_2, yielding,

```
L2: do i=2,n
L1:  do j=2,n
      a(i,j) = a(i-1,j-1)+b(i)
     enddo
    enddo
```

which would satisfy Rule 4.9, yielding the original loop nest. The process would continue ad infinitum.

[4] Recall that only < directions can be associated with levels that carry dependencies.

Scalar Expansion

Some recurrences cannot be moved to outer loops. Perhaps there is no outer loop to move the recurrence to or the conditions of Rule 4.9 are not satisfied. In these circumstances, scalar expansion [49] can sometimes be applied to eliminate a recurrence.

The idea behind scalar expansion is simple. Recall from Chapter 2 that output and anti dependencies arise from the re-use of variables. With the scalar expansion operator, we provide distinct variables for every use by expanding scalars into arrays. The drawback is, of course, more memory usage. The benefit, however, might be faster execution.

As an example, consider the following loop.

```
    do i=1,100
S1: x = a(i)+b(i)
S2: c(i) = x*2
    enddo
```

The dependencies on x, $S_1\,\delta^+\,S_2$ and $S_2\,\delta_c\,S_1$, form a recurrence that inhibits the vectorization of S_1 and S_2. A scalar expansion of x into the new array %x, yields the following loop.

```
    do i=1,100
S1: %x(i) = a(i)+b(i)
S2: c(i) = %x(i)*2
    enddo
```

Both S_1 and S_2 can now be vectorize.

The scalar expansion operator is represented in $\mathcal{M}$ with the class `scalar-expand` whose attributes are:

- `var` is the name of variable to expand.
- `stmt` is the statement's tag.
- `reason` is the reason for the expansion.
- `wrt-loop` is the loop to expand with respect to.

As with loop interchange, scalar expansion is attempted only when it results in changing a recurrence that directly inhibits the vectorization of a statement. The following rule adds an element of the class `scalar-expand` under the appropriate conditions.

Rule 4.10

```
(rule scalar-expansion
  (vectorizable ^stmt <st>)
  (statement ^stmt <st> ^depth <d>)
  (loop ^stmt <loop> ^depth <d>)
  (recurrence-member ^stmt <st> ^id <id> ^carrier <loop>)
  (recurrence ^id <id> ^from <st1> ^to {<st2> <> <st1>} ^vname <v>)
  (dependence ^var <v> ^type anti ^ref VAR_REF ^carrier <loop>)
  -(dependence ^var <v> ^type true ^extent loop-carried)
  -(scalar-expand ^var <v>)
-->
  (make scalar-expand ^var <v> ^wrt-loop <loop>
     ^reason break-then-vectorize ^stmt <st>))
```

The first five condition elements require that a vectorizable statement be a member of a recurrence and that the dependence causing the recurrence is a scalar anti dependence carried by the inner loop. The next condition element requires that there be no loop-carried true dependencies on the same variable. If there are true dependencies, a more complex transformation involving partial loop unrolling is required.

4.4 Parallelizing Loops

As mentioned in Chapter 1, some machines have vector hardware, some have parallel hardware, some have both. In this section, we focus on the parallel execution of loop iterations. This is a common form of parallelism and is supported by most parallel dialects of the FORTRAN language. To simplify the discussion, only the `doall` [29] style constructs that impose no explicit synchronization between loop iterations are considered. Other constructs, such as `doaccross` [29], that allow synchronization are not discussed.

For semantic invariance the parallelize transformation operator requires, among other things, that no loop-carried dependencies exist. The reason for this should be clear. Given two iterations of a loop assigned to separate processors, absent synchronization, no guarantee can be made about the execution order of the iterations. If a loop-carried dependence exists, then a semantically required order is imposed on the execution of the loop iterations. Without synchronization, the required order may be violated.

There are other considerations for parallelized loops. The semantics of I/O operations are usually undefined if executed in parallel. For our discussion, if the loop body contains I/O statements, the loop is not eligible for parallelization.

Likewise, the semantics of parallel function evaluation is usually undefined. Thus, loop bodies that contain function references are similarly ineligible for parallelization.

The following rule is similar to Rule 4.5. It marks statements as parallelizable if they do no I/O and contain no function references.

Rule 4.11

```
(rule parallelizable
  (parallelize-phase)
  (parallel-capable)
  {<ce> (loop-member ^stmt <st> ^parallelizable no)}
  -(function-reference ^stmt <st>)
  -(statement ^stmt <st> ^class << WRITE FORMAT READ >>)
```

```
 -->
   (modify <ce> ^parallelizable yes))
```

If all statements in the loop are marked as parallelizable and the loop is not a carrier of any dependence, then the loop can be parallelized.

Rule 4.12
```
(rule parallelize-loop
  (loop ^id <id> ^stmt <loop>)
  -(loop-member ^id <id> ^parallelizable no)
  -(dependence ^extent loop-carried ^carrier <loop>)
-->
  (make parallelize-loop ^id <id> ^stmt <loop>))
```

4.4.1 Localization

As with vectorization, when parallelizing a loop, we require that the loop not be a carrier of a dependence. A principle similar to scalar expansion, known as scalar localization, can be used to break loop-carried dependencies.

Many dialects of FORTRAN support local declarations whose scope is the body of a loop, but whose extent, semantically at least, is just one iteration of the loop. When iterations of a loop are distributed among the available processors, a separate copy of each loop local variable is allocated on each processor. Since each iteration of the loop has a local variable, the vectorizer's job of expanding a scalar into an array is a simple matter of local variable declaration for the parallelizer. The implementation (language and hardware) handles the dirty work for the parallelizer.

This is the companion rule for scalar expansion. We localize scalars that are loop-carried. If there are no loop-carried dependencies and all members of the loop are parallelizable, then the loop can be parallelized.

Rule 4.13
```
(rule localize-scalar
  (loop ^id <loop-id> ^stmt <loop>)
```

```
   -(loop-member ^id <id> ^parallelizable no)
   (dependence ^var <v> ^extent loop-carried ^type << output anti >>
      ^carrier <loop> ^ref VAR_REF)
   -(dependence ^var <v> ^extent loop-carried ^type true ^carrier <loop>)
   -(localize-scalar ^var <v> ^stmt <loop>)
 -->
   (make localize-scalar ^var <v> ^stmt <loop>))
```

4.5 Summary

In this chapter we have discussed the transformation operators, their preconditions, and the representation of the associated program and machine properties. In the next chapter, we extend the ideas presented here by presenting a set of rules for updating $\mathcal{M}$, planning, backtracking, and refining plans.

Chapter 5

Planning

The previous chapter introduced several program transformation operators along with a set of rules to select operators. In this chapter we examine the issues involved in planning a sequence of transformation operators. We develop a planning model and present a set of rules, based on the operators from Chapter 4, that implement the model.

5.1 Introduction

Program restructuring is a process that involves planning a sequence of transformation operators to affect some performance improvement while at the same time maintaining the semantic properties of the program. In this chapter we examine the issues involved in finding a sequence of transformation operators that guarantee semantic invariance (validity constraint) and improve performance (usefulness constraint).

Planning a sequence of operators can be difficult. By definition, transformation operators change the program in some way. These changes often interact. Such interaction can cause an otherwise appropriate operator to violate one or both of the restructuring constraints.

In the next section we present a hierarchical planning model for a rule-base restructurer. In this model, the planner attempts to avoid foreseeable operator conflicts while constructing a crude plan. The crude plan is examined and repaired by a higher

level planner. If a crude plan is unsalvageable, the planner backtracks, constructing a new plan that is again subject to evaluation.

A critical feature of this model is that the planner maintains a consistent description of the transformed program through the use of update rules associated with each operator. For each operator, the update rules make the appropriate changes to $\mathcal{M}$ so that $\mathcal{M}$ represents the program properties present after the application of the operator.

5.2 Planning Model

The interactions between operators make planning a sequence of operators a non-trivial task. In the field of artificial intelligence, planning methods have received a considerable amount of attention. Hierarchical strategies have proven useful in domains where operator interaction is significant [60].

Hierarchical planning involves developing plans of successively higher resolution at successively higher levels. The final plan, derived from the highest planning level, has the desired resolution. From this perspective, hierarchical planning is an iterative process in which initial, crude plan is repaired or refined in successive phases until a plan emerges that is free of defects.

In this section, we present a hierarchical planning model in which successive phases focus on refining the program/machine match. The initial, crude plan developed from property-directed rules of Chapter 4 is refined by the operator-directed refinement rules presented below.

The planning model presented here is based primarily on empirical results and experience in program restructuring. The approach we take differs somewhat in that no hierarchy of abstraction spaces (which are usually associated with hierarchical planners) are employed. Instead, we focus on a hierarchy of conditions. This condition

hierarchy stratifies the operator selection process into managable levels.

The hierarchical planning model is characterized by the following features.

Condition Hierarchy Associated with each level in the hierarchy is a set of conditions that are significant to the level. Conditions outside a given level are viewed as details to be worked out at other levels.

Property-Directed Low-Level Planning Operator selection is based on program and machine properties. The rules of Chapter 4 are used to select basic transformation operators.

Operator-Directed High-Level Refinement Operator selection is based on the operators entered into the plan at lower levels. For example, assume that a `vectorize` operator appears in the plan. During high-level plan refinement, an operator might be introduced to harvest the loop in sections so as to reduce the impact of startup overhead or improve memory performance.

$\mathcal{M}$ Consistency Associated with each operator are rules to update $\mathcal{M}$ so that it remains consistent with the transformed state of the program.

Backtracking Fatally flawed, crude plans are discarded. The offending operators are inhibited in the next attempt. A new crude plan is constructed and submitted for refinement.

5.2.1 Condition Hierarchy

At the lowest levels, the rules of Chapter 4 are used to select the transformation operators, thus creating the initial, crude plan. Using other transformation operators such as statement ordering, strip mining, and loop fission, successive levels refine the crude plan.

Condition	Level	Operators/Property
ASSIGN	0	vectorizable
ARRAY_REF	0	vectorizable
no function references	0	vectorizable
vectorization not inhibited	0	vectorizable
loop length and stride ok	0	vectorizable
vectorizable	1	vectorize, vectorize-with-guard
vectorizable	1	scalar-expand, interchange-loops
no inner level recurrence	1	vectorize, vectorize-with-guard
no IF control dependence	1	vectorize
IF control dependence	1	vectorize-with-guard
scalar recurrence	1	scalar-expand
inner level recurrence	1	scalar-expand, interchange-loops
legal loop interchange possible	1	interchange-loops
no function reference	2	parallelizable
no I/O performed	2	parallelizable
parallelizable	2	parallelize-loop
no inner loop-carried	2	parallelize-loop
inner loop-carried	2	localize-scalar
vectorize	3	loop-fission
better strip size	4	strip-mine, interchange-loops
need new statement order	5	re-order statements
similar, small loops	6	loop-fusion

Table 3: Condition Hierarchy

The conditions at each level direct the application of the rules at the level. The consideration of other conditions, such as statement order, optimal vector size, and optimal strip size, is postponed until higher levels. This basic organization is repeated at each level. Associated with each level is a set of conditions to which the rules of the level are sensitive. Table 3 illustrates the stratification of conditions by level and operator or property derived.

Organizing the conditions into a hierarchical form allows the rule-based reasoning system to focus on a simplified set of priorities. When the rules of a level are

exhausted, the next level, together with its set of conditions, is used to guide the reasoning process.

5.2.2 Property-Directed Low-Level Planning

We divide the levels in the planning hierarchy into two groups; the property-directed low-level group and the operator-directed high-level group.

The low-level group uses the rules of Chapter 4 to select transformation operators based on the properties presented by or derived from the program. The derived properties include the dependence relations discussed in Chapter 2 and the recurrence relations discussed in Chapter 3. Other properties presented by the program include the control dependencies, statement classifications, and textual ordering.

5.2.3 Operator-Directed High-Level Plan Refinement

In operator-directed high-level group, operators are selected based on the operator selections made during property-directed low-level planning. Operators selected during this phase improve the efficacy of the low-level operators or remove debilitating interactions between low-level operators.

Statement Re-Ordering

As discussed in Chapter 3, the dependence graph, in general, contains cycles. Partitioning the graph into its strongly connected components (π-blocks) and sorting topologically yields an ordering consistent with the semantics of the program. Rule 3.12 together with the rules that derive the π-blocks, generate the new ordering. But why is a new ordering required? The following example illustrates a case where the original statement ordering together with a valid set of statement transformation operations violates the original program semantics.

```
    do i=1,n
S1: a(i) = b(i)
S2: b(i+1) = k(i)
    enddo
```

Vectorizing statements S_1 and S_2, yields,

```
S1: a(1:n) = b(1:n)
S2: b(2:n+1) = k(1:n)
```

which has a meaning different from the loop.

In the loop, values in array `b` determine the values in the array `a`. In turn, values in array `k` determine values in array `b`. In the vectorized statements, however, array `a` is determined by the values in `b` *prior* to any reference to `k`. Hence, the ordering of the vectorized statements has a different meaning. In the loop there is a true dependence from S_2 to S_1. This dependence is unchanged by the vectorization process. A topological sort of the π-blocks[1] yields the correct statement ordering.

```
S2: b(2:n+1) = k(1:n)
S1: a(1:n) = b(1:n)
```

Statement re-ordering is done after low-level transformation operators have been selected. The low-level operators result not only in vectorization and parallelization, but may break recurrences and generate new π-blocks. Often, this new collection of π-blocks must be re-ordered to maintain semantic invariance.

[1]Since there are no cycles in the dependence graph for this loop, each statement is in a separate π-block.

Loop Fission

When vectorization occurs to some, but not all, statements in a loop, the vectorized statements must be removed from the innermost nesting of the loop. Because of dependencies between statements in the loop and the vectorized statements, this might involve *fissioning* the loops. Consider the following loops.

```
    do i=1,n
S1: a(i) = cos(b(i))**2
S2: c(i) = d(i)+e(i)*k
S3: write(6,2) c(i)
    enddo
```

The conditions of Rule 4.5 permit the vectorization of statement S_2, but do not permit the vectorization of S_1 or S_3. Loop fission results in two loops.

```
    do i=1,n
S1: a(i) = cos(b(i))**2
    enddo
S2: c(1:n) = d(1:n)+e(1:n)*k
    do i=1,n
S3: write(6,2) c(i)
    enddo
```

Loop Fusion

Loop fusion is the inverse of loop fission. Because loop overhead can be significant, successive loops that have identical bounds are merged into a single loop.

Statement re-ordering applied to the previous example yields:

```
S2: c(1:n) = d(1:n)+e(1:n)*k
    do i=1,n
S1: a(i) = cos(b(i))**2
```

```
    enddo
    do i=1,n
S3: write(6,2) c(i)
    enddo
```

This is a prime candidate for loop fusion.

```
S2: c(1:n) = d(1:n)+e(1:n)*k
    do i=1,n
S1:  a(i) = cos(b(i))**2
S3:  write(6,2) c(i)
    enddo
```

Since loop fusion is the inverse of loop fission, any system that supports both must be careful not to fuse the same loops previously fissioned.

Strip Mining

Strip mining [53], sometimes called loop sectioning or loop blocking, divides the iteration space of a loop into blocks of fixed size. Each block can either be vectorized or parallelized. The size of a block corresponds to either the optimum vector size or number of processors of the target machine.

For a machine with 64 element vector registers, the loop,

```
do i=1,256
 a(i) = b(i) + c(i)
enddo
```

would be mined in 64 element blocks.

```
do j=0,3
 do i=j*64+1,(j+1)*64
```

```
    a(i) = b(i) + c(i)
   enddo
  enddo
```

Strip mining is useful for vector machines where there is a limited vector size. While most vector machines have a fairly small maximum vector size, some do not. The Cyber-205, a memory-to-memory architecture, is an example of a machine with the rather large maximum vector length of 64k.

Strip mining has also been shown to be useful in reducing the demand on memory caches [36]. By blocking a loop into sections small enough to fit into the cache, the locality of reference of a loop can be strengthened, thus increasing the number of cache hits and improving performance. This strategy has also been used to improve performance of virtual memory systems [1].

Rule 5.1

```
(rule strip-mine
  (vectorize ^stmt <st>)
  (statement ^stmt <st> ^depth <d>)
  (loop-member ^id <id> ^stmt <st>)
  (loop ^id <id> ^depth <d>)
  (vector-size ^best <size>)
  -(strip-mine ^id <id>)
-->
  (make strip-mine ^id <id> ^size <size>))
```

5.2.4 $\mathcal{M}$ Consistency

The representation of program properties as well as the rules of the previous chapters were designed specifically to allow for the effects of transformation operators. The rules of this section carry out the necessary updates to $\mathcal{M}$.

Loop Interchange

When loops are interchanged, the direction vectors associated with dependencies carried by the loops must be changed to reflect the new loop nesting. Loop interchange moves a recurrence carrier loop from the innermost nesting level to an outer nesting level. Unless this change is reflected in the direction vector representation, the loop interchange will not have its intended effect.

To update $\mathcal{M}$ after a loop interchange, the corresponding elements of the direction vectors are interchanged.

Rule 5.2

```
(rule loop-interchange-change-directions
  (interchange-loops ^this <d1> ^that <d2> ^request-id <c>)
  {<ce0> (direction ^dir <dir0> ^depth <d1> ^dim <dim> ^request-id <> <c>
     ^var-from <vt1> ^var-to <vt2>)}
  {<ce1> (direction ^dir <dir1> ^depth <d2> ^dim <dim> ^request-id <> <c>
     ^var-from <vt1> ^var-to <vt2>)}
-->
  (modify <ce0> ^dir <dir1> ^request-id <c>)
  (modify <ce1> ^dir <dir0> ^request-id <c>))
```

A `loop-interchange` operator together with each pair of corresponding direction vector elements satisfies this rule. The direction vector elements are interchanged and marked (via `request-id`). The marking guarantees that the rule is satisfied only once for each pair of direction vectors.

Scalar Expansion

A fundamental difference between scalar expansion and loop interchange is that with scalar expansion, recurrences are removed. With loop interchange, recurrences are displaced to other loop levels. Also, unlike loop interchange, scalar expand results in the introduction of a new array. References to this new array might provide new sources for vectorization. Both of these differences are reflected in the way $\mathcal{M}$ is updated after the introduction of a scalar expand operator.

Scalar expansion changes scalar references to array references. This introduction of array references might reveal new sources for vectorization. The following rule modifies all references to the variable from scalar reference to an array reference. Assuming other factors are present, Rule 4.5 will use this change to assert that the referring statement is vectorizable.

Rule 5.3

```
(rule scalar-expand-change-references
  (scalar-expand ^var <v>)
  {<ce> (reference ^var <v> ^var-type VAR_REF)}
-->
  (modify <ce> ^var-type ARRAY_REF))
```

The purpose of scalar expansion is to remove an inhibiting recurrence. Once the recurrence is removed, the members of the recurrence become candidates for vectorization. The following rules delete the loop-carried dependencies and the recurrences associated with the scalar. The rules also update the π-blocks.

Rule 5.4

```
(rule scalar-expand-delete-dependencies
  (scalar-expand ^var <v>)
  {<ce> (dependence ^var <v> ^extent loop-carried)}
-->
  (remove <ce>))
```

Scalar expansion breaks a recurrence by converting a loop-carried scalar dependence into a loop-independent dependence (cf. Theorem 3.1). Since the associated recurrences are deleted and the affected π-blocks are updated, failure to delete these dependencies could result in a cyclic π-block graph. Cycles in the π-block graph result in a non-terminating topological sort. For each loop-carried dependence involving the expanded scalar, Rule 5.4 deletes each loop-carried dependence involving the expanded scalar.

The following pair of rules work together to delete the recurrences that are based on loop-carried dependencies involving the expanded scalar. The first of these rules removes the `recurrence` class and generates a subgoal to remove the members of the class. The second rule removes the members.

Rule 5.5

```
(rule scalar-expand-delete-recurrences
  (scalar-expand ^var <v>)
  {<ce> (recurrence ^id <id> ^vname <v>)}
-->
  (remove <ce>)
  (make remove-recurrence-member ^id <id>))
```

Rule 5.6

```
(rule delete-recurrence-members
  (remove-recurrence-member ^id <id>)
  {<ce> (recurrence-member ^id <id> ^stmt <st>)}
-->
  (make remove-pi-block-member ^stmt <st>)
  (remove <ce>))
```

To keep $\mathcal{M}$ consistent, in altering the recurrences represented in $\mathcal{M}$ the π-blocks represented must also be altered. Notice that for each recurrence member removed, Rule 5.6 generates a subgoal to remove the associated π-block member. Since π-block members (statements) may be members of more than one recurrence, they cannot simply be removed from a π-block when one of the recurrences is deleted. However, when all of the recurrences that contain a particular statement are deleted by rules 5.5 and 5.6, then the statement must be removed from a π-block. One solution uses a reference count in each element of the class `pi-block-member`. The reference count is decremented each time Rule 5.7 is satisfied.

Rule 5.7

```
(rule dec-pi-block-member
  {<ce1> (remove-pi-block-member ^stmt <st>)}
```

```
  {<ce2> (pi-block-member ^stmt <st> ^cnt <c>)}
-->
  (remove <ce1>)
  (modify <ce2> ^cnt (- <c> 1)))
```

When the reference count reaches zero, the π-block member is removed from $\mathcal{M}$.

Rule 5.8

```
(rule delete-pi-block-member
  {<ce> (pi-block-member ^cnt <= 0)}
-->
  (remove <ce>))
```

When a π-block contains has no members, it is removed from $\mathcal{M}$ by the following rule.

Rule 5.9

```
(rule delete-pi-blocks
  {<ce> (pi-block ^id <id>)}
  -(pi-block-member ^id <id>)
-->
  (remove <ce>))
```

Finally, every statement must be a member of exactly one π-block if the topological sort of the π-blocks is to yield a correct statement ordering. The actions of the previous rules may orphan some statements. The following rule creates a new π-block for each orphaned statement.

Rule 5.10

```
(rule orphans
  (statement ^stmt <st>)
  -(pi-block-member ^stmt <st>)
-->
  (let ([<id> (gensym)])
   (make pi-block ^id <id> ^seq (set!  *sequence* (+ 1 *sequence*)))
   (make pi-block-member ^id <id> ^stmt <st>)))
```

Scalar Localization

As with scalar expansion, scalar localization side-effects $\mathcal{M}$. However, scalar localization is used in conjunction with loop parallelization, not vectorization. Loop parallelization requires that no loop-carried dependencies exist. Membership in recurrences is not considered by Rule 4.12. Although by Theorem 3.1 the former implies the latter, by explicitly requiring in Rule 4.12 that no loop-carried dependencies exist, the lengthy update of recurrences and π-blocks need not be done. The removal of loop-carried dependencies is still required and is accomplished by the following rule.

Rule 5.11

```
(rule localize-scalar-delete-dependencies
  (localize-scalar ^var <v> ^stmt <loop>)
  {<ce> (dependence ^var <v> ^extent loop-carried ^carrier <loop>)}
-->
  (remove <ce>))
```

5.2.5 Backtracking

Some plans contain operators that do not achieve their intended goal. As long as these operators do not change the state of $\mathcal{M}$ in such a way as to inhibit other goal-satisfying operators, they represent little more than a loss of efficiency. However, when the unneeded operators causes changes in $\mathcal{M}$ that might inhibit other operators, the plan is considered fatally flawed. A new plan must be created that does not include the unneeded operators. Discarding all or a part of a plan and retreating to a previous choice point the search tree is called *backtracking*.

Since there is no simple way to return to a previous $\mathcal{M}$ state, backtracking is expensive in our planning model. Essentially, backtracking requires that $\mathcal{M}$ be rebuilt using the rules of Chapters 2, 3, and 4. In Chapter 6, we discuss ways that $\mathcal{M}$ can be partitioned so that only parts need to rebuilt. Even with this enhancement, backtracking has a non-trivial complexity that makes it undesirable.

What sort of situation forces the planner to backtrack and rebuild $\mathcal{M}$? Consider the following loop.

```
L1: do j=2,n
L2:  do i=2,n
S1:   a(i,j) = a(i-1,j)+b(i-1)+x
S2:   x = x*2
S3:   c(i) = d(i)*k
     enddo
    enddo
```

Statement S_1 is classified as `vectorizable` by Rule 4.5, however, the statement cannot be vectorized because it is a member of a recurrence carried by loop L_2. Interchanging loops L_1 and L_2 would move the recurrence to the outer loop. But, S_1 still cannot be vectorized because of the loop-carried scalar true dependence, $S_2\,\delta_c^t\,S_1$, forms a recurrence carried by the inner loop.[2] The problem is that interchanging L_1 and L_2 does not remove *all* of the recurrences that S_1 is involved in. Ultimately, the plan contains no `vectorize` operators, but does contain a single, unnecessary `loop-interchange` operator. By making L_2 the outermost loop, the vectorization of S_3 is prevented. The plan is fatally flawed.

The backtracking solution is to rebuild $\mathcal{M}$, explicitly inhibit the vectorization of S_1 (cf. the last condition element of Rule 4.5), and restart the planner. This time, a `interchange-loop` operator is not proposed for S_1, hence the conditions are satisfied for the vectorization of S_3.

The need to backtrack after low-level plan development is easily recognized by rules of the following form.

[2] Recall from Chapter 4 that scalar expansion is not used to break recurrences carried by a scalar true dependence.

Rule 5.12

```
(rule unnecessary-loop-interchange-first
   (check-fatal-flaws)
   (interchange-loops ^reason vectorize ^stmt <st>)
   -(vectorize ^stmt <st>)
   -(backtrack-needed)
-->
   (make backtrack-needed)
   (in-ps planner-specialist (make inhibit-vectorization ^stmt <st>)))
```

The meaning of the condition elements should be obvious. The right hand side contains the `in-ps` special form. More will be said about `in-ps` special form in Chapter 6.

5.3 Examples

In this section we take a detailed look at the operator selection and planning process with two examples.

The first example illustrates the use of scalar expansion to remove recurrences that block vectorization. In the following loop, there is a loop-carried scalar anti dependence from S_2 to S_1.

```
     do i=1,n
S1:  x = a(i)+b(i)
S2:  c(i) = x*2
     enddo
```

The rules of Chapters 2 and 3 build the dependence graph, recurrences, and π-blocks. These elements, together with elements derived from a parse of the program, are used by the rules of this chapter and Chapter 4.

Assuming that the target machine is capable of vector processing, Rule 4.5 marks statement S_1 as vectorizable. The resulting change to $\mathcal{M}$ together with the presence of the scalar recurrence causes Rule 4.10 to be satisfied. Rule 4.10 enters the

`scalar-expand` operator to the plan.

The use of `scalar-expand` in the plan causes the $\mathcal{M}$ consistency rules described above to become active. Rule 5.3 changes the references to `x` from `VAR_REF` to `ARRAY_REF`, thus enabling the vectorization of S_2 as well. Rules 5.4, 5.5, 5.6, and 5.7 delete the loop-carried dependence and the recurrence, and decrement the π-block reference count. This enables Rule 4.6 to add the `vectorize` operator for both S_1 and S_2 to the plan.

At this point, the plan is to scalar expand `x` and vectorize S_1 and S_2. By harvesting the loop in sections of the optimal size (see Section 4.2), the match between the code and the underlying hardware can be further improved. During plan refinement, Rule 5. 1 adds the `strip-mine` operator to the plan.

Assuming the optimal vector size is 64 and `n` is a multiple of 64, execution of this final plan results in the following code.

```
    do i=1,n,64
S1: %x(i:i+64) = a(i:i+64)+b(i:i+64)
S2: c(i:i+64) = %x(i:i+64)*2
    enddo
```

By exploiting both the vector capabilities of the hardware and the optimal vector size, this version of the loop better matches the target machine. The essential point in the plan development process was the removal of the recurrence. The need to remove the recurrence is detected by the Rule 4.10. Once the recurrence has been removed and the associated updates made to $\mathcal{M}$, other operators are chosen.

In the next example `loop-interchange` is used to move the carrier loop of a recurrence to the outermost position.

```
L1: do 20 j=1,n
L2:  do 21 i=2,n
S:    a(i,j) = a(i-1,j) + b(i)
     enddo
    enddo
```

Statement S cannot be vectorized because of the recurrence characterized by the loop-carried true dependence, $S\,\delta_1^t\,S$. The direction vectors, `(=,<)` and `(=,=)`, suggest that interchanging loop L_2 and loop L_1 would move the recurrence to outermost level, thus enabling the vectorization of S. Rule 4.9 is satisfied by these conditions and adds the `interchange-loops` operator to the plan. The resulting changes to $\mathcal{M}$ satisfy the conditions for vectorization (see Rule 4.6).

5.4 Summary

This chapter discusses a simple model for the selection and planning of a sequence of transformation operators. Operator interaction is reduced, although not eliminated, by employing a hierarchical planning strategy. Successive levels in the hierarchy represent successively better plan refinement. Some conditions require that a plan developed at a lower level be abandoned and a new plan proposed. Such backtracking, although expensive, is required.

Chapter 6

Organization of a Rule-Based Restructurer

The previous chapters discussed the methods of a rule-based program restructurer. In this chapter we discuss some issues involved in the organization of a rule-based program restructurer. We assume that the restructurer uses the planning model of Chapter 5, the dependence, recurrence, and π-block graph construction rules of chapters 2 and 3, and the operator selection rules of Chapter 4.

6.1 Introduction

As mentioned in Chapter 1, an experimental rule-based restructurer, called REX, serves as a testbed for the results discussed in the previous chapters. In this chapter we continue this theme beyond the scope of conditions and operators to the somewhat introspective realm of the reasoning system's organization. The purpose of this chapter is not to describe the details of a particular rule-based restructurer, but to discuss the issues involved in building a restructurer that implements the results of the previous chapters. Where appropriate, we use REX as an example rule-based restructurer.

6.2 Control

The data-driven nature of rule-based reasoning makes control a complex and sometimes a troublesome issue in rule-based reasoning systems. Our planning model requires a good deal of control. Control is complicated by the interactions between rules. In fact, it has been suggested that a rule-based system could not be used in a program restructurer because of the complexity of rule interactions [68].

6.2.1 Lexicographic Conflict Resolution

Rules become active based on the contents of $\mathcal{M}$. However, it is not the case that only one rule is satisfied by $\mathcal{M}$ at any particular time. In fact, under most circumstances, a number of rules are satisfied simultaneously. Selecting a particular rule from the *conflict set* is performed by a process called *conflict resolution*. Because conflict resolution plays a critical part in defining the semantics of a rule-based reasoning system, it is discussed in some detail here.

Many conflict resolution strategies have been proposed and used in rule-based reasoning systems [54]. Rather than review other methods, in this section we describe the conflict resolution strategy assumed for the rules presented in this thesis.

The inference process of a rule-based reasoning system consists of cycle of three phases: 1) rule satisfaction, 2) conflict resolution, and 3) rule application.

In the rule satisfaction phase, a state saving discrimination network, called a RETE Tree [34], is used to determine the set of *instantiations.*[1] These instantiations are entered into the conflict set. Conflict resolution is the process of selecting from the conflict set, a single instantiation for the rule application phase. After the rule application phase, the process continues with the rule satisfaction phase. Only one rule is selected from the conflict set on each cycle.

[1]An instantiation is a rule together with the subset of $\mathcal{M}$ that satisfies the rule.

Refraction is the final step in conflict resolution. Refraction is the removal of the selected instantiation from the conflict set. Without removing the instantiation from the conflict set, the next iteration of the cycle might select the instantiation again, causing the rule to become active again with the same subset of elements from $\mathcal{M}$. Refraction guarantees that the rule will become active only once for each satisfying subset of $\mathcal{M}$.

A unique time stamp is associated with each element in $\mathcal{M}$. The time stamp, which denotes an element's recency, imposes a partial order on the elements of $\mathcal{M}$. In the first step of conflict resolution, the instantiations in the conflict set are ordered lexicographically by the time stamps of the elements matching the first condition, the second, the third, and so on. If one instantiation is ordered before all others, it becomes the selected instantiation. It is removed from the conflict set (refraction) and passed to the rule application phase.

If two or more rules are lexicographically equivalent, then the instantiation with the most complex left hand side is selected. Complexity is computed by counting both the number of condition elements and the number of predicates on the left hand side. If two or more of the remaining instantiations have the same complexity, then an arbitrary instantiation is selected, removed, and passed to the rule application phase.

Notice that with this conflict resolution strategy, rule order plays no part in the selection of an instantiation from the conflict set. The strategy uses only the recency of elements, and failing that, the complexity of the condition elements.

In building a rule-based system, the recency can be used to control rule selection. Recency is used to force the inference system to focus on sorting π-blocks in Rule 3.12. But using recency to enforce some desired control structure has its limitations. It would be difficult, for example, to use recency to enforce a broad control structure over a large, complex set of rules.

In the next section, we discuss an alternative method of imposing a control structure on a rule-based reasoning system.

6.2.2 Selecting Levels With Context Elements

Recency helps insure that a particular subgoal will get continued attention by the inference process, but it cannot, in general, be applied to a complex set of goals. In this section, we discuss a method of enabling some subset of rules by the use of *context elements* [20].

The idea behind context elements is simple. Suppose in some phase of the reasoning process, we need to focus the inference process on some sub-collection of rules. We can limit rule satisfiability to this sub-collection by adding to all the rules in the sub-collection, a condition element that is only satisfied during the phase.

Many of the rules of Chapter 4 used these context elements to limit satisfiability to a particular level corresponding to the condition hierarchy discussed in Chapter 5. For example, consider Rule 4.5

Rule 4.5

```
 (rule vectorizable
   (vector-capable)
   (vectorize-phase)
   (statement ^class ASSIGN ^stmt <st> ^depth <d>)
   (reference ^stmt <st> ^var-type ARRAY_REF)
   (loop ^id <id> ^depth <d>)
   (loop-profile ^id <id> ^length ok ^stride ok)
   -(function-reference ^stmt <st>)
   -(vectorizable ^stmt <st>)
   -(inhibit-vectorization ^stmt <st>)
 -->
   (make vectorizable ^stmt <st>))
```

Here, the context must be `vectorize-phase` for Rule 4.5 to be satisfiable by $\mathcal{M}$. This is an effective and widely used method for focusing the inference process. However, using context elements requires that the context elements be added to the

left hand sides of the individual rules. This limits the flexibility because the rules are *only* satisfiable in the given context. Under some circumstances, we might want a sub-collection of rules to be satisfiable in several contexts.

In addition to limiting flexibility, context elements do not help deal with backtracking. Essentially, backtracking requires that $\mathcal{M}$ be purged, operator inhibiting elements added, and the inference process restarted. Clearly this is expensive. Some of the cost can be saved if only *part* of $\mathcal{M}$ is purged. Conceivably, this we could have rules that determine which elements of $\mathcal{M}$ should be removed. But in the final analysis, this added overhead is likely to outweigh any benefits reaped from partially purging $\mathcal{M}$. In the next section, we discuss a new method of encapsulating entire sub-collections of rules and subsets of $\mathcal{M}$.

6.2.3 Rule Systems as First-Class Objects

We define a rule system as a collection of rules together with a set of elements, $\mathcal{M}$, a conflict set, and an environment. The rules of the system are only sensitive to the set of elements, $\mathcal{M}$, associated with the system. The only instantiations in the conflict set associated with the system are those derived from satisfaction of rules of the system by elements from the memory $\mathcal{M}$, associated with the rule system.

Definition 6.1 *A rule system is a tuple of the form, $< R, \mathcal{M}, C, E >$, where R is a collection of rules, $\mathcal{M}$ is a set of elements, C is a set of instantiations derived from the satisfaction of members of R by subsets of $\mathcal{M}$, and E is an environment.*

In REX, a rule system is a procedural, first-class object. It is procedural in the sense that it can be applied.[2] It is first-class in the sense that it can be passed to

[2]When applied to a single non-negative integer argument, the rule system performs the indicated number of satisfy-select-act cycles. Without an argument, the rule system cycles until the conflict set is empty. All other arguments are undefined.

Specialist	**Function**
`top-level`	Handles high-level plan refinement
`dependence`	Constructs DDG and δ^+ relations
`recurrence`	Builds recurrence and π-block relations
`planner`	Handles low-level plan development
`marker`	Provides user-interface marking support

Table 4: Specialists organization of a rule-based restructurer.

and returned from procedures, bound to variables, and stored in data structures.[3] As with other objects in SCHEME, the extent of rule system objects is indefinite.

All of the procedures and special forms of REX associated with rule-based programming effect the lexically closest rule system. As one would expect, all the scoping rules of SCHEME apply to rule systems.

Specialists

The ability to encapsulate rule systems as first-class objects allows a restructurer to be organized as a collection of separate rule systems. By separating the rule space into groups, we can significantly reduce the overhead involved in backtracking, while at the same time providing another level of control beyond recency and context elements.

Experiments with REX suggest that at least five rule systems, called *specialists*, are useful (see Table 4). Each system contains the rules specific to its domain. The `dependence` specialist, for example, uses the rules of Chapter 2 to derive the data dependence graph along with the δ^+ relations. Likewise, the `recurrence` specialist uses the rules of chapter 3 and 4 to derive the recurrence and π-block relations.

How does this help backtracking? Let us assume that during plan refinement, the `top-level` specialist determines that a plan submitted by the `planner` is fatally

[3] Including elements of *any* $\mathcal{M}$.

flawed. The `top-level` specialist can initiate actions that purge the memory of the `planner`, inhibit the offending parts of the plan, and restart the `planner`. Notice that the work done by the `dependence` and `recurrence` specialists is still valid—only the plan created by the `planner` needs to be rebuilt. Not only does this provide considerable computational savings, but it also allows the `top-level` specialist to maintain record of previous troublesome operators.

Communication

One of the problems with the specialist organization is communication between rule systems. Since memory spaces are disjoint, we need a new special form to transmit the changes made in one rule system to the memory of another rule system.

The `in-ps` special form takes two arguments. The first argument is a rule system object. The second is an expression. In the uniprocessor model, the expression evaluated in an environment containing the given production system. Thus, changes made to memory by the expression are reflected in the given production system. In a distributed model, changes could be encapsulated into messages which are sent to the rule systems over some interconnection network.

The `in-ps` form is too low level a mechanism to perform the services needed in backtracking. We can, however, use the `in-ps` form in a set of *copy rules* for each specialists.

Essentially, a copy rule has two preconditions: a production system to copy to and an element from $\mathcal{M}$ to copy. The right hand side uses the `in-ps` form to make the change in the foreign rule system's memory. A copy rule in the `dependence` specialist to copy a dependence edge to another specialist is of the form,

Rule 6.2

```
(rule copy-dependencies
  (copy-to <ps>)
  (dependence ^var <v> ^from <st1> ^to <st2> ^type <type>
```

```
      ^ref <ref> ^extent {<extent> <> nil}
      ^carrier <carrier> ^vfrom <vt1> ^vto <vt2>)
 -->
   (in-ps <ps>
    (make dependence ^var <v> ^from <st1> ^to <st2> ^type <type>
      ^ref <ref> ^extent <extent>
      ^carrier <carrier> ^vfrom <vt1> ^vto <vt2>)))
```

6.3 Support Routines

In addition to the rule-based environment, a rule-based restructurer similar to the one described in this thesis needs access to at least three other resources: a language parser (FORTRAN in this case), a subscript analyzer, and a user interface.

The language parser builds a parse tree representation of the program under study from the source code. Using information obtained by a parse of the program along with simple flow analysis, the specialists construct individual abstract program representations. These representations drive the restructuring process.

Although rule-based methods for subscript analysis have been proposed [19], for simplicity we assumed that subscript analysis is performed by a separate GCD Test-based solver.

Finally, an interactive interface is useful. The user interface should display the source code, the current restructuring plan, and possibly allow input from the user.

6.4 Summary

In this chapter we have provided a brief overview of an organization for a rule-based program restructurer. We discussed the issue of control from three aspects: recency, context elements, and specialists.

Chapter 7

Thesis Summary

In this chapter we present a brief overview of the results of the thesis. We also suggest avenues for future research.

7.1 Overview of Results

In Chapter 1, we defined the problem under study and presented the two requirements for program restructuring: validity and usefulness. We discussed Flynn's characterization of computer architectures and related this characterization to the exploitable sources of parallelism. We presented our view of program restructuring as one of improving the match between program level and the hardware level and presented some examples of program restructuring. Finally, we outlined the goals of the thesis and reviewed the related research.

In Chapter 2, we discussed the theory of data dependence analysis and derived one well known test, the GCD Test. The GCD Test is used in the rules developed in Chapter 2 to analyze dependence equations that arise during subscript analysis. We stated and proved the Commutativity Theorem and defined three data dependence relations, δ^t, δ^a, and δ^o. We developed a representational scheme for the dependence relations, variables, use-definition and definition-use chains, and direction vectors. This scheme is used to represent the data dependence graph in $\mathcal{M}$. We developed six rules that derive the three dependence relations, δ^t, δ^a, and δ^o. These rules also

distinguish between loop-carried and loop-independent dependencies.

In Chapter 3, we defined the transitive closure relation, δ^+, and developed a set of rules to derive the transitive closure of dependence relations in the dependence graph. Next, we stated and proved a theorem that characterized certain dependence chains as recurrences in the dependence graph. We used this theorem to develop rules to find recurrences in the graph. Finally, in Chapter 3 we developed a series of rules to partition the dependence graph into strongly connected components. We presented a rule that sorts these components, called π-blocks, topologically. This topological sort yields a new, valid statement ordering.

In Chapter 4, we discussed the program and machine properties that affect the program restructuring process. We introduced several rules to derive machine properties. We developed rules that select both vectorization and parallelization operators. Along with these rules we also developed rules for removing operator inhibiting program properties such as inner loop recurrences.

In Chapter 5, we developed a hierarchical planning model. The model consists of a condition hierarchy, property-directed low-level planning, operator-directed high-level planning, methods for updating $\mathcal{M}$, and a strategy for backtracking. We introduced a series of high-level operators that either improve memory performance or improve the efficacy of an operator selected at a lower level in the planning hierarchy. Finally, we presented a series of examples that illustrate the planning model.

In Chapter 6, we discussed the organization of a rule-based program restructurer. We concentrated on the issue of control and presented three ways to modify control in a rule-based reasoning system. We introduced a new approach to control in which collections of rules are bundled into first-class objects. We showed how this approach simplifies backtracking.

7.2 Future Work

This thesis, of course, does not explore every aspect of rule-based program restructuring. Many avenues of research in this field remain to be investigated. In what follows, we present just a few of the open problems and areas for future work.

In the remarks at the end of Section 2.4.2, we noted that the representational set chosen as the starting point for dependence analysis limited what we could reason about. Specifically, we noted that by not representing execution paths we could not reason about variable aliases.

The variable alias problem can be stated as follows. Let $S_1 \; \delta \; S_2$ hold because of some variable v. Further, assume that variable v' references the same memory location as v. Variable v' is said to be an *alias* for v. Aliases can come from the use of `equivalence` statements or `common` blocks in FORTRAN. For many other languages, pointers provide another mechanism for variable aliasing.

Whatever the source, variable aliases present obvious problems for dependence analysis. Brandes [19] has investigated rule-based methods of dependence analysis capable of reasoning about execution paths. Combining these methods with the results from Chapter 2, might be fruitful in addressing the variable aliasing problem.

The dependence analysis of Chapter 2 was limited to a single region of the program. Procedures invoked in this region where implicitly assumed to not modify any of the dependence relations. In most circumstances, however, shared, global variables or modified procedure parameters could result in a different set of dependencies. Interprocedural dependence analysis [67] attempts to find these other sources of dependencies by examining the entire program, including all of the procedures. This analysis is more complex than the analysis presented in Chapter 2, but the methods of Chapter 2 could be extended to include an interprocedural analysis component.

In the literature one finds many more transformation operators than presented in

Chapter 4. Additional rules could be added to accommodate these other transformations. This would require additional rules to implement each of the transformations along with rules to update $\mathcal{M}$ to reflect the application of the operators. Additional condition levels may also be required in the planning model.

The results of this thesis could be extended to work in areas outside of program restructurering. For example, the current generation of reduced instruction set computers (RISC) [57] need very sophisticated compilers [25, 41, 23]. The compiler for a RISC architecture must determine strategies for handling branch prediction, register allocation, pipeline scheduling, and many other issues. These problems could be addressed through the use of a rule-based reasoning system. Some of the results of this thesis, especially those of chapters 2 and 3, could be used to improve a RISC compiler.

Appendix A

Rex-UPSL Syntax

REX-UPSL is a programming language primarily designed for building large systems of program restructuring rules, for historical reasons, called production systems. The left hand side patterns for rules in REX-UPSL are written using a syntax very similar to OPS5 [33]. The right hand side actions of rules are written in SCHEME [59]. In this appendix we present an extended Backus-Naur Form (BNF) syntax for REX-UPSL. The syntax for SCHEME expressions is adapted from the offical SCHEME report [59] and is integrated into the sytactic description of REX-UPSL where appropriate.

An in depth discussion of the language and its uses can be found in [64].

The BNF presented here is extended with the following notation.

- Terminals are written in **boldface type**
- Descriptive terminals are written in *italics*.
- Items followed by + occur one or more times.
- Items followed by * occur zero or more times.

A.1 BNF Syntax

program	$\longrightarrow$	statement*
statement	$\longrightarrow$	rule | expression | declaration
declaration	$\longrightarrow$	(**literal** lit-dcl+)

| (**literalize** symbol symbol+)
| (**vector-attribute** symbol+)

system ⟶ (**ps** statement+) | (**named-ps** symbol statement+)
lit-dcl ⟶ symbol = number
tl-ce ⟶ (symbol tl-lhs-term*)
tl-lhs-term ⟶ ^ symbol tl-lhs-value | ^ number tl-lhs-value
tl-lhs-value ⟶ symbol | num
rule ⟶ (**rule** symbol lhs --> rhs)
lhs ⟶ positive-ce ce*
ce ⟶ positive-ce | negative-ce
positive-ce ⟶ form | { element-var form } | { form element-var }
negative-ce ⟶ - form
form ⟶ (symbol lhs-term*)
lhs-term ⟶ ^ symbol lhs-value | ^ number lhs-value | lhs-value
lhs-value ⟶ { restriction*} | restriction
restriction ⟶ << any-atom* >> | predicate atomic-value | atomic-value
atomic-value ⟶ ' symbol | var-or-const
var-or-const ⟶ symbol | number | variable
predicate ⟶ <> | = | < | <= | >= | > | <=>
rhs ⟶ expression
token ⟶ identifier | boolean | number | character | string
| (|) | #(| ' | .
deliminator ⟶ whitespace | (|) | " | ;
whitespace ⟶ *space, newline, or tab*
comment ⟶ ; *characters up to newline*
atomsphere ⟶ whitespace | comment
intertoken-space ⟶ atomsphere*

identifier ⟶ initial subsequent∗ | perculiar-identifier

initial ⟶ letter | special-initial

letter ⟶ **a** | **b** | . . . | **Z**

special-initial ⟶ **!** | **$** | **%** | **&** | ***** | **/** | **:** | **<** | **=>** | **?** | **~** | **_** | **^**

subsequent ⟶ initial | digit | special-subsequent

digit ⟶ **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

special-subsequent ⟶ **.** | **+** | **-**

expresison-keyword ⟶ **quote** | **lambda** | **if** | **set!** | **begin** | **cond**
| **and** | **or** | **let** | **let*** | **letrec**
| **literalize** | **vector-attribute**

variable ⟶ *any identifier that is not a keyword*

boolean ⟶ **# t** | **# f**

character ⟶ #\any-character

any-character ⟶ *any character*

number ⟶ + unumber | - unumber

unumber ⟶ digit digit∗| digit . digit∗

string ⟶ " any-character∗"

symbol ⟶ identifier

datum ⟶ simple-datum | compound-datum

simple-datum ⟶ boolean | number | character | string | symbol

compound-datum ⟶ list | vector

list ⟶ (datum∗)| (datum datum∗. datum)| ' datum

vector ⟶ #(datum∗)

expression ⟶ variable | literal | procedure-call | lambda-expr
| conditional | assignment | derived-expr | system

literal ⟶ quotation | self-evaluating

self-evaluating ⟶ boolean | number | character | string

quotation → ' datum | (**quote** datam)

procedure-call → (operator operand*)

operator → expression

operand → expression

lambda-expr → (**lambda** formals expression*)

formals → (variable*) | variable | (variable variable*. variable)

conditional → (**if** expression expression expression)

assignment → (**set!** variable expression)

derived-expression → (**cond** cond-clause+)

| (**cond** cond-clause*(**else expression+**))

| (**and** expression*)

| (**or** expression*)

| (**let** (binding-spec) expression*)

| (**let*** (binding-spec) expression*)

| (**letrec** (binding-spec) expression*)

| (**begin** expression*)

cond-clause → (expression*)

binding-spec → (variable expression)

Appendix B

Element Attributes

This appendix contains a complete list of all the element attributes used in the thesis. Although some of the information presented here is duplicated in the body of the thesis, this appendix brings together all of the element attribute declarations in one place.

B.1 Element Attribute Declarations

The `declare-attribute` special form is used to associate element attributes with element classes. The symbol following the `declare-attribute` is the class name. All other symbols are attributes of the class.

B.2 Element Attributes

```
(declare-attribute dependence
                   var            ; variable name
                   from           ; statement from
                   to             ; statement to
                   type           ; true, anti, output
                   ref            ; ARRAY_REF or VAR_REF
                   extent         ; loop-carried or loop-independent
                   carrier        ; loop that carries the dependence
                   vfrom          ; variable instance from
                   vto)           ; variable instance to
```

```
(declare-attribute direction
                   dir            ; direction at this depth
                   dim            ; array dimension
                   depth          ; depth
                   carrier-depth  ; depth that carries the dependence
                   var-from       ; variable instance from
                   var-to         ; variable instance to
                   request-id)    ; used in update


(declare-attribute recurrence
                   id             ; unique recurrence identifier
                   from           ; instance of loop-carried
                   to             ; dependence that caused recurrence
                   members        ; list of members
                   vname)         ; variable name


(declare-attribute recurrence-member
                   id             ; unique recurrence identifier
                   stmt           ; statement
                   carrier)       ; carrier loop for recurrence


(declare-attribute statement
                   class          ; statement
                   stmt           ; statement class
                   depth          ; loop nesting level for statement
                   control        ; control parent
                   order)         ; relative statment order


(declare-attribute function-reference
                   function       ; function name
                   stmt)          ; statement


(declare-attribute reference
                   var            ; variable name
                   stmt           ; statement
                   var-type)      ; ARRAY_REF or VAR_REF
```

```
(declare-attribute pi-block
                   id             ; unique pi-block identifier
                   seq)           ; sequence number


(declare-attribute pi-block-member
                   id             ; unique pi-block identifier
                   stmt           ; statement
                   cnt)           ; number of recurrences referenced


(declare-attribute vectorizable
                   stmt)          ; statement


(declare-attribute vectorize
                   stmt)          ; statement


(declare-attribute vectorize-with-guard
                   guard-var      ; guard variable
                   stmt           ; statement to vectorize
                   guard-stmt)    ; if statement


(declare-attribute vectorize-if
                   guard-var      ; guard variable
                   stmt)          ; if statement to vectorize


(declare-attribute scalar-expand
                   var            ; variable to expand
                   stmt           ; statement
                   reason         ; reason for scalar expansion
                   wrt-loop)      ; loop to expand with respect to
```

```
(declare-attribute interchange-loops
                   id             ; id of recurrence
                   stmt           ; statement affected
                   reason         ; reason for interchange
                   this           ; this loop
                   that           ; that loop
                   request-id     ; used in direction vector update
                   inner          ; new inner loop statement
                   outer)         ; new outer loop statement


(declare-attribute next-pi-block
                   seq)           ; next pi-block to generate


(declare-attribute gather-pi-block-members
                   id             ; id of pi-block to gather
                   members)       ; members of the pi-block


(declare-attribute inhibit-vectorization
                   stmt)          ; statement


(declare-attribute localize-scalar
                   var            ; variable to localize
                   stmt)          ; loop statement


(declare-attribute parallelize-loop
                   id             ; loop id
                   stmt)          ; loop statement


(declare-attribute order
                   id             ; pi-block identifier
                   seq)           ; sequence
```

```
(declare-attribute remove-recurrence-member
                   id)           ; recurrence members to remove


(declare-attribute remove-pi-block-member
                   stmt)         ; statment to remove


(declare-attribute region
                   body)         ; region of code to analyze


(declare-attribute loop
                   id            ; loop id
                   stmt          ; loop header statement
                   depth         ; depth of this loop
                   length        ; upper - lower
                   lower         ; lower bound
                   upper         ; upper bound
                   request-id    ; used in update
                   step          ; step value
                   induction-var) ; induction variable


(declare-attribute loop-member
                   id            ; loop id
                   stmt          ; statement that is a member of loop
                   parallelizable); yes or no


(declare-attribute loop-profile
                   id            ; loop id
                   stride        ; is stride acceptable?
                   length)       ; is loop length acceptable?


(declare-attribute vector-length
                   min           ; min useful vector length
                   max           ; max allowable vector length
                   best)         ; best vector length
```

```
(declare-attribute var
                   var            ; variable name
                   stmt           ; statement
                   access         ; use or def
                   instance       ; variable instance
                   ref)           ; ARRAY_REF or VAR_REF


(declare-attribute flow
                   var            ; variable name
                   from           ; statement from
                   to             ; statement to
                   flow-type)     ; use-def, def-def, or def-use


(declare-attribute dependence+
                   path           ; statements along path from..to
                   from           ; statement from
                   to)            ; statement to
```

Bibliography

[1] Abu-Sufah, W., Kuck, D., and Lawrie, D. Automatic Program Transformations for Virtual Memory Computers. In *Proc. National Computer Conference* (June 1979), pp. 969–974.

[2] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers Principles, Techniques, and Tools.* Addison Wesley, 1985.

[3] Allen, F., Burke, M., Charles, P., Cytron, R., and Ferrante, J. An Overview of the PTRAN Analysis System for Multiprocessing. *Journal of Parallel and Distributed Computing 5* (March 1988), 617–640.

[4] Allen, F. E., and Cocke, J. A Catalogue of Optimizing Transformations. In *Design an optimization of compilers* (March 1971), R. Rustin, Ed., Prentice-Hall, pp. 1–30.

[5] Allen, J., and Kennedy, K. Conversion of Control Dependence to Data Dependence. In *10th ACM Symposium on Principles of Programming Languages* (1983), ACM, pp. 177–189.

[6] Allen, J. R., and Kennedy, K. Automatic Loop Interchange. In *SIGPLAN Notices: Proc. ACM SIGPLAN 84 Sym. on Compiler Construction* (June 1984), vol. 19.

[7] Allen, R., Baumgartner, D., Kennedy, K., and Porterfield, A. PTOOL: A Semi-automatic Parallel Programming Assistant. Tech. rep., Department of Computer Science, Rice University, January 1987.

[8] Allen, R., and Kennedy, K. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. on Programming Languages and Systems 9*, 4 (October 1987), 491–542.

[9] Balasundarum, V., Kennedy, K., Kremer, U., and McKinley, K. The ParaScope Editor: An Interactive Parallel Programming Tool. In *Proc. of the Supercomputing 89* (November 1989), pp. 540–550.

[10] Banerjee, U. Data Dependence in Ordinary Programs. Master's thesis, University of Illinios at Urbana-Champaign, 1976.

[11] Banerjee, U. A Theory of Loop Permutations. In *Languages and Compilers for Parallel Computing* (1989), MIT Press, pp. 54–74.

[12] Banerjee, U., Chen, S.-C., Kuck, D., and Towel, R. Time and Parallel Processor Bonds for FORTRAN-like Loops. *IEEE Trans. Comput. C-28*, 9 (1979).

[13] Bernstein, A. Analysis of Programs for Parallel Processing. *IEEE Trans. Electronic Computers 15* (1966), 757–62.

[14] Bodin, F., Windheiser, D., Jalby, W., Atapattu, D., Lee, M., and Gannon, D. Performance Evaluation and Prediction for Parallel Algorithms on the BBN GP1000. Tech. Rep. 304, Computer Science Department, Indiana University, February 1990.

[15] Bose, P. Heuristic Rule-based Transformations for Enhanced Vectorization. In *1988 International Conference on Parallel Processing* (1988), pp. 63–66.

[16] Bose, P. Interactive Program Improvement via EAVE: An Expert Advisor for Vectorization. In *Proc. 1988 ACM Int'l. Conf. on Supercomuting* (January 1988).

[17] Brainerd, W. S., Goldberg, C. H., and Adams, J. C. *Programmer's Guide to* Fortran 90. McGraw Hill, 1990.

[18] Brandes, T. Automatic Vectorisation for High Level Languages Based on an Expert System. In *Lecture Notes in Computer Science no. 237: CONPAR86* (1986), pp. 303–310.

[19] Brandes, T. Determination of Dependencies in a Knowledge-Based Parallelization Tool. *Parallel Computing 8* (1988), 111–119.

[20] Brownston, L., Farrel, R., Kant, E., and Martin, N. *Programming Expert Systems in OPS5.* Addison-Wesley, 1986.

[21] Byler, M., Davies, J. R. B., Huson, C., Leasure, B., and Wolfe, M. Multiple Version Loops. In *Proceedings of the International Conference on Parallel Processing* (August 1987), pp. 312–318.

[22] Callahan, D., Cocke, J., and Kennedy, K. Estimating Interlock and Improving Balance for Pipelined Architectures. *Journal of Parallel and Distributed Computing*, 5 (1988), 334–358.

[23] Chow, F., Himelstein, M., Killian, E., and Weber, L. Engineering a RISC Compiler System. In *Proceedings of COMPCON Spring 86* (March 1986), pp. 132–137.

[24] Cohagen, W. L. Vector Optimization for the ASC. In *Proceedings of the Seventh Annual Princeton Conference on Information Sciences and Systems* (Princeton, N.J., 1973), Dept. of Electrical Engineering, pp. 169–174.

[25] Coutant, D., Hammond, C., and Kelly, J. Compiler for the New Generation of Hewlett-Packard Computers. In *Proceedings of the COMPCON Spring 86* (March 1986), pp. 48–61.

[26] Crowther, W., Goodhue, J., Starr, E., Thomas, R., Milliken, W., and Blackadar, T. Performance Measurements on a 128-node Butterfly Parallel Processor. In *Proc. International Conference on Parallel Processing* (August 1985).

[27] Date, C. *An Introduction to Database Systems.* Addison-Wesley, 1983.

[28] Davies, J., Huson, C., Macke, T., leasure, B., and Wolfe, M. The KAP/S-1: An Advanced Source-to-Source Vectorizer for the S-1 Mark IIa Supercomputer. In *Proc. International Conference on Parallel Processing* (1986), pp. 833–835.

[29] Eigenmann, R., Hoeflinger, J., Jaxon, G., Li, Z., and Padua, D. Restructuring fortran Programs for Cedar. In *Proc. of the International Conf. on Parallel Processing* (1991), pp. 57–66.

[30] Eigenmann, R., Hoeflinger, J., Jaxon, G., and Padua, D. Cedar fortran and Its Compiler. In *Lecture Notes in Computer Science, No. 457: CONPAR90-VAPP IV* (1990), pp. 288–299.

[31] Flynn, M. Very High Speed Computing Systems. *Proc. IEEE 54* (1966), 1901–1909.

[32] Flynn, M. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers C-21*, 9 (September 1972).

[33] Forgy, C. OPS5 User's Manual. Tech. Rep. CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, july 1981.

[34] Forgy, C. RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence 19* (1982), 17–37.

[35] GANNON, D., GUARNA, V. A., AND LEE, J. K. Static Analysis and Runtime Support for Parallel Execution of C. In *Languages and Compilers for Parallel Computing* (1989), D. Gelernter, A. Nicolau, and D. Padua, Eds., MIT Press, pp. 254–275.

[36] GANNON, D., JALBY, W., AND GALLIVAN, K. Strategies for Cache and Local Memory Management by Global Program Transformation. *Journal of parallel and distributed computing 5* (1988), 587–616.

[37] GERNDT, H. M., AND ZIMA, H. P. Optimizing Communication in Superb. In *Lecture Notes in Computer Science No. 457: CONPAR90-VAPP IV* (1990), pp. 300–311.

[38] GOTTLIEB, A., GRISHMAN, R., KRUSKAL, C. P., MCAULIFFE, K. P., RUDOLPH, L., AND SNIR, M. The NYU Ultracomputer – Desiging a MIMD Shared Memory Parallel Computer. *IEEE Trans. on Computers* (February 1983), 175–189.

[39] HAGHIGHAT, M. R. Symbolic Dependence Analysis for High Performance Parallelizing Compilers. Tech. Rep. 995, CSRD, May 1990.

[40] HIGBEE, L. Vectorization and Conversion of FORTRAN Programs for the Cray-1 CFT Compiler. Tech. Rep. 2240207, Cray Research Inc., Mendota Heights, Minn., June 1979.

[41] HOPKINS, M. Compiling for the RT PC ROMP. Tech. Rep. SA23-1057, IBM, 1986.

[42] HORD, R. M. *Parallel Supercomputing in SIMD Architectures.* CRC Press, 1990.

[43] Huson, C., Macke, T., Davies, J. R., Wolfe, M., and Leasure, B. The KAP/205: An Advanced Source-to-Source Vectorizer for the Cyber 205 Supercomputer. In *Proceedings of the 1986 International Conf. on Parallel Processing* (1986), IEEE Computer Society Press, pp. 827–832.

[44] Kremer, U., Bast, H.-J., Gerndt, M., and Zima, H. Advanced Tools and Techniques for Automatic Parallelization. *Parallel Computing*, 7 (July 1988), 387–393.

[45] Kuck, D. *The Structure of Computers and Computations.* John Wiley, 1978.

[46] Kuck, D., Kuhn, R., Padua, D., Leasure, B., and Wolfe, M. Dependence Graphs and Compiler Optimizations. In *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages* (1981), pp. 207–218.

[47] Kuck, D., Muraoka, Y., and Chen, S.-C. On the Number of Operations Simultaneously Executable in FORTRAN-like Programs and Their Resulting Speedup. *IEEE Trans. Comput. C-21* (1972), 1293–1310.

[48] Kuck, D. J. ILLIAC IV Software and Application Programming. *IEEE Trans. on Computers C-17*, 8 (August 1968), 758–770.

[49] Kuck, D. J., Sameb, A. H., Cytron, R., Veidenbaum, A. V., Polychronopoulos, C. D., Lee, G., McDaniel, T., Leasure, B. R., Beckman, C., Davies, J. R. B., and Kruskal, C. P. The Effects of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance. In *1984 International Conference on Parallel Processing* (1984), R. M. Keller, Ed., pp. 129–138.

[50] LAMPORT, L. The Parallel Execution of DO Loops. *Comm. of ACM 17* (1974), 83–93.

[51] LEE, M.-H. *Data Localization in Parallel Computer Systems.* PhD thesis, Dept. of Computer Science, Indiana University, February 1991. Techinal Report No. 325.

[52] LEVESQUE, J. M. Applications of the Vectorizer for Effective Use of High-Speed Computers. In *High Speed Computer and Algorithm Organization*, D. Kuck, D. Lawrie, and A. H. Sameh, Eds. Academic Press, New York, 1977, pp. 447–449.

[53] LOVEMAN, D. B. Program Improvement by Source-to-Source Transformation. *JACM 24*, 1 (1977), 121–145.

[54] MCDERMOTT, J., AND FORGY, C. Production System Conflict Resolution Strategies. In *Pattern-Directed Inference Systems*, D. Waterman and F. Haynes-Roth, Eds. Academic Press, Inc, 1978.

[55] MYSZEWSKI, M. The Vectorizer Systems: Current and Proposed Capabilites. Tech. Rep. CA-17809-1511, Massachusetts Computer Associates, Inc., Wakefield, MA, September 1978.

[56] PADUA, D. A., AND WOLFE, M. J. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM 29*, 12 (December 1986).

[57] PATTERSON, D. A. Reduced Instruction Set Computers. *Communications of the ACM 28*, 1 (January 1985), 8–21.

[58] RAMAMOORTHY, C., AND GONZALEZ, M. A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs. In *1969 Joint Comput. Conf., AFIPS Conf. Proc.* (1969), vol. 35, AFIPS Press, pp. 1–15.

[59] REES, J., AND CLINGER, W. Revised[3] Report on the Algorithmic Language Scheme. Tech. Rep. 174, Department of Computer Science, Indiana University, December 1986.

[60] SACERDOTI, E. Planning in a Hierarchy of Abstraction Spaces. *Artificial Intelligence 5*, 2, 115–135.

[61] SMITH, K. PAT – An Interative FORTRAN Parallelizing Assistant Tool. In *1988 International Conference on Parallel Processing* (1988), pp. 58–62.

[62] TANENBAUM, A. S. *Structured Computer Organization.* Prentice-Hall, 1984.

[63] TARJAN, R. Depth First Search and Linear Graph Algorithms. *SIAM J. Comput. 1*, 2 (1972), 146–160.

[64] TENNY, L. UPSL User's Manual. Tech. Rep. 257, Department of Computer Science, Indiana University, Bloomington, Indiana, 1988.

[65] TORRES, J., AYGUADE, E., LABARTA, J., LLABERIA, J., AND VALERO, M. On Automatic Loop Data-mapping for Distributed-memory Multiprocessors. In *Lecture Notes in Computer Science no. 487* (1991), pp. 173–182.

[66] TRELEAVEN, P. Control-driven Data-driven, and Demand-driven Computer Architecture. *Parallel Computing 2* (1985).

[67] TRIOLET, R. Interprocedural Analysis for Program Restructuring with Parafrase. Tech. Rep. 538, Center for Supercomputer Research and Development, Univ. of Illinios, Urbana, 1985.

[68] WANG, K.-Y. *Intelligent Program Optimization and Parallelization for Parallel Computers.* PhD thesis, Purdue University, May 1991.

[69] WANG, K.-Y., AND GANNON, D. Applying AI Techniques to Program Optimization for Parallel Computers. In *Parallel Processing for Supercomputers and Artificial Intelligence* (1989), K. Hwang and D. DeGroot, Eds., McGraw-Hill, pp. 441–485.

[70] WEDEL, D. FORTRAN for the Texas Instruments ASC System. *ACM SIGPLAN Notices 3*, 10 (March 1975).

[71] WOLFE, M. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, 1982.

[72] WOLFE, M. Advanced Loop Interchanging. In *Proc. of the 1986 International Conference on Parallel Processing* (1986), IEEE Computer Science Press, pp. 536–543.

[73] WOLFE, M. Vector Optimization vs Vectorization. *Journal of Parallel and Distributed Computing 5* (1988), 551–567.

[74] WOLFE, M. The Tiny Loop Restructuring Research Tool. In *Proc. of the 1991 International Conference on Parallel Processing* (1991), pp. 46–53.

[75] ZIMA, H. *Supercompilers for Parallel and Vector Computers.* Frontier Series. ACM Press, 1990.

[76] ZIMA, H., BAST, H.-J., AND GERNDT, M. SUPERB: A Tool for Semiautomatic MIMD/SIMD Parallelization. *Parallel Computing 6* (June 1988), 1–18.