Compaction-Based Parallelization

Alexander Aiken
Ph.D. Thesis

TR 88-922
June 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

COMPACTION-BASED PARALLELIZATION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Alexander Scott Aiken

August 1988

COMPACTION-BASED PARALLELIZATION

Alexander Scott Aiken, Ph.D.

Cornell University 1988

We present a transformational system for extracting parallelism from programs. Our transformations generate code for synchronous parallel computers, such as Very Long Instruction Word and pipelined machines.

The transformational system, which is based on percolation scheduling, is simple and uniform. There are four primitive transformations—three that perform code motion plus loop unrolling—from which all parallelizing algorithms are constructed.

Our transformations are studied as a formal system. We define a formal measure of program improvement, and show that our transformations improve programs with respect to the measure. This formal approach allows a number of results on the expressive power of our transformations. Most importantly, we show that it is possible to compute limits of infinite sequences of the primitive transformations. This leads to a number of new algorithms for software pipelining, including: an algorithm that generates optimal code for loops without tests, an algorithm for software pipelining of multiple nested loops, and a general solution to the problem of software pipelining in the presence of tests.

Using the four primitives and the limit-taking transformation, it is possible to express the classical parallelization techniques for vector, multiprocessor, and VLIW machines, such as doacross, the wavefront method, loop interchange, trace scheduling, and a simple form of vectorization. Thus, our transformational system can be viewed as a formal foundation for the area of parallelization.

Biographical Sketch

Alex Aiken was born in Stockholm, Sweden, on March 5, 1964. He grew up and attended college in Ohio, graduating from Bowling Green State University in 1983 with a Bachelor of Science in Music and Computer Science. After completing his undergraduate work, he spent a year in Salzburg, Austria, studying German and trombone. Midway through that year, he realized that his assets (about $45) were insufficient to fund a return trip to the United States to audition for music graduate school. Reasoning that this was the best excuse he was likely to find for leaving professional music, Alex applied to and entered the Computer Science Ph.D. program at Cornell in 1984. Things proceeded smoothly from there; he received a Masters of Science in Computer Science in January, 1987 and his Ph.D. in August, 1988.

Acknowledgements

Many people have assisted me in the research and writing of this dissertation. My advisor, Alex Nicolau, introduced me to parallelization and spent countless hours discussing problems. Prakash Panangaden answered my questions about semantics and guided me out of several dead-ends. Ed Murray served as the minor member of my committee. Anne Neirynck met with me for months to discuss the problems of providing formalisms for program transformations. Stanley Trepetin ran many of the experiments on the system. Micah Beck wrote a software package that allows pictures to be incorporated into LaTeX, saving me hours of cutting and pasting. Dexter Kozen suggested a nice proof of Lemma 8.3.5. And, finally, thanks to Jennifer Widom, for everything else.

Table of Contents

List of Figures

Chapter 1

Introduction

Parallel computers promise to deliver dramatic increases in speed over sequential machines. However, while it is obvious that a parallel computer has the potential for higher performance, actually realizing this potential is difficult. Programming parallel machines has long been recognized to be a considerably more time-consuming, complex, and error-prone task than programming sequential machines [Gil58].[1] One promising solution to this problem is *parallelization*: techniques for automatically translating programs into parallel machine code.

We study parallelization for tightly-coupled synchronous parallel machines, such as Very Long Instruction Word (VLIW) and pipelined machines. Many commercial machines fall into this class, including horizontally microcoded engines, the Mars-432 [Num83], FPS-164/264 [Cha81], Multiflow's TRACE series [Mul87], and Cydrome's Cydra [Cyd87].[2] These machines have several distinguishing features. There is a single flow of control (i.e., a single program counter) and a global address space. Parallelism is achieved by multiple (synchronous) functional units. Finally, these machines are statically scheduled—virtually all

[1]For an interesting historical example of the difficulty in the context of programming multiprocessors, see [Dij65,Hym66,Knu66].

[2]TRACE, Multiflow, and Trace Scheduling are registered trademarks of Multiflow Computer. Cydra and Cydrome are registered trademarks of Cydrome Inc.

runtime behavior is specified by the program.

We consider parallelization techniques based on the *data dependencies* of a program. Data dependencies represent ordering constraints on the execution of program statements that must be preserved. (Dependency analysis is discussed in detail below.) This is a natural restriction—most of the important parallelization techniques in the literature are based on dependency analysis.

We focus on the parallelization of scientific (numerical) programs. While our basic methods are applicable to any class of programs, we address some problems specific to the parallelization of scientific programs.

Our approach to parallelization is based on *program transformations*. A program transformation is a function that takes a program and produces a semantically equivalent program that is improved in some way. The transformational system we develop is simple and uniform. There are four primitive transformations from which all parallelizing algorithms are constructed. Our transformations fall into the general class of *compaction* transformations. In the compaction paradigm, parallelism is extracted from a program by scheduling statements in the same time step to shorten (compact) the paths through a program, thereby making the program faster.

We use a formal model to study our transformational system. The model captures the important aspects of synchronous parallel computation and data dependency analysis. This formal approach leads to new results on the power of compaction-based parallelization as well as new parallelization algorithms. Most interestingly, we show that it is possible to compute limits of infinite sequences of the primitive transformations. This limit-taking transformation allows our system to perform *software pipelining*—a parallelization technique for overlapping the iterations of a loop in much the same way that a hardware pipeline overlaps a stream of instructions. We develop new algorithms for software pipelining and present several general theoretical results, including a characterization of all software pipelining algorithms.

Although parallelization has been an active area of research for more than twenty years, there is little understanding of the theoretical properties of parallelization or of the relationships between the various parallelization techniques proposed in the literature. We show that the classical parallelization techniques for other classes of parallel machines are also limits of infinite sequences of our primitive transformations. These results establish a precise theoretical relationship between our compaction-based system and other approaches and show that the effect of these techniques can be realized by a compaction-based compiler.

1.1 Systems of Program Transformations

Many constraints make designing a good system of program transformations difficult. Ideally, a system of program transformations should meet all of the following criteria:

- The transformations should be correct—they should preserve program semantics. While this seems obvious, some standard transformations do alter the semantics of a program.

- The transformations should improve the program in some way. This is also not as self-evident as it may seem; some common optimizations actually decrease the performance of a program in some circumstances [ASU86].

- The transformational system should be expressive. That is, it should be possible to significantly improve the program using the transformations.

- Testing the conditions required for a transformation should be inexpensive. We term this property the *applicability* of a transformation. A transformational system is not useful if it is difficult to decide when transformations apply.

- Algorithms built from the transformations should be efficient.

There is no absolute standard for judging a transformational system. A particular transformation may be very good for one class of machines and wholly inadequate for another. Thus, the properties of a given system must be judged with respect to a given model. In our case, the model is a class of synchronous parallel machines.

We illustrate the difficulty of designing a good transformational system with two simple examples. Let the model be some machine M and let the measure of improvement be execution time. A transformation from program P to program P' is written $P \mapsto P'$. Consider the transformational system $\mathcal{T}$:

$$\mathcal{T} = \left\{ P \mapsto P' | P' \text{ is the fastest program for } M \text{ equivalent to } P \right\}$$

Clearly this is a very expressive system and all transformations are correct and improve the program. However, testing the conditions for a transformation is hard—even determining what the transformations are is undecidable [Rog87]. Now consider the transformational system $\mathcal{T}'$:

$$\mathcal{T}' = \{ P \mapsto P | P \text{ is a program} \}$$

In this system, the transformations are correct, they satisfy at least a weak measure of improvement (nothing gets worse), and testing the condition for a transformation is trivial. However, $\mathcal{T}'$ is not an expressive system.

These extreme examples illustrate two points. First, there are limits on the capabilities of a compiler. Computing the fastest program equivalent to a given program is undecidable, so no practical transformational system can be expressive using this standard. Thus, program transformations are studied in restricted contexts. In our case, we consider transformations that use only data dependency information. Second, in general there are unavoidable trade-offs amongst the criteria listed above. For example, $\mathcal{T}$ and $\mathcal{T}'$ show that it is impossible to achieve both optimal expressive power and optimal applicability for model M. The best one can hope for is an expressive system with reasonable computational requirements. The model is also a parameter—within reason, we are free to modify the model as we choose. One of the motivations behind the development of RISC architectures was a desire to build a machine (a model) well-suited to the expressive power of compiler technology [Pat85].

An important corollary of our second point is that powerful transformations do not necessarily make a powerful system. As a general rule, the coarser a transformation is—the more it accomplishes in a single step—the more restrictive the conditions are under which a compiler can actually apply the transformation. ($\mathcal{T}$, for example, is a system composed of extremely coarse transformations.) Furthermore, coarse transformations are inflexible—relatively few of all possible parallelizations of a program can be achieved by a finite system of coarse transformations. In contrast, our transformations are "fine"; one application of any of the transformations has only a small effect on the execution of the program. The entire system of transformations, however, is flexible and provably expressive.

1.2 A Short History of Parallelization

Parallelization was first achieved by hardware techniques. The IBM System/360 Model 91, for example, used a sophisticated hardware algorithm to overlap instructions within basic blocks [AST67].[3] Subsequently, compiler techniques for parallelization were developed. Bernstein was the first to address the problem automatically detecting parallelism in programs [Ber66]. Three early parallelizing compilers were based directly on Bernstein's work [Rus69,RG69,BF67]. (Of these, it appears that only the system of Ramamoorthy and Gonzalez [RG69] was ever described in published journals.) A survey of early approaches to parallelization and an extensive bibliography are in [Bae73].

[3] A basic block is a program fragment where no statement, except possibly the last, is a branch.

```
for i := 1 to N do
        A[i] := A[i] + 1;
```

(a) Original program.

```
A[1 : N] := A[1 : N] + 1;
```

(b) Vectorized program.

Figure 1.1: A simple example of vectorization.

```
for i := 1 to N do
    for j := 1 to N do
        A[i, j] := A[i, j] + 1;
```

(a) Original program.

```
on processor i do
    for j := 1 to N do
        A[i, j] := A[i, j] + 1;
```

(b) Outer loop parallelized.

Figure 1.2: A simple example of parallelization for a multiprocessor.

Initially, there was considerable interest in blending programming language features and compiler technology for parallelization; see, for example, [BBK+68, Kuc68,RTT73,RT74]. This interest continues today (e.g., in defining extensions to Fortran [For86]), but most current research concentrates either on parallel programming languages (such as Parallel Pascal [Ree84] and Joyce [BH87]) or parallelizing compilers.

Parallelization has been extensively studied for three basic machine models. Historically, the first model was the vector machine. As the name suggests, vector machines are suited to the repetitive matrix/vector computations found in scientific programs. A vector machine is usually a scalar machine with an attached high-speed, pipelined floating-point unit. (For convenience, we also classify array processors as vector machines.) The vector processor is the classic SIMD (Single Instruction Multiple Data [Fly66]) architecture. The floating-point unit applies the same instruction to many different data. *Vectorization* is the basic parallelization technique for producing code for vector machines. Roughly, vectorization looks for repetitive calculations on arrays and converts them to vector instructions. A simple example is given in Figure 1.1; the compiler detects that all the additions are independent and generates one vector instruction for the entire loop.

The second basic machine model is the asynchronous multiprocessor. Parallelization for multiprocessors differs considerably from parallelization for vector machines. Interprocessor communication costs are generally the dominant factor in a multiprocessor program. To reduce communication, parallelizing algorithms partition the program relatively coarsely, attempting to give each processor a large, communication-free task to execute. An example of multiprocessor parallelization is given in Figure 1.2. In this example, all the iterations of the outer loop are parallelized; each inner loop is assigned to a single processor. The third basic model is the tightly-coupled synchronous machine discussed above. Both multiprocessors and tightly-coupled synchronous machines are MIMD (Multiple Instruction Multiple Data) architectures; at any moment different instructions may be executed using different data.

In a number of publications, the *Parafrase* project under Kuck at the University of Illinois has defined many of the concepts in parallelization for vector machines and multiprocessors [Mur71,KMC72,KM75,Tow76,Kuc76,Ban76, Kuc78,Ban79,PH79,Kuh80,KKLW80,KKP+81,Wol82,Cyt84]. Kennedy and the members of the *PFC* project at Rice University have also done important work in compilation for these machines [AK82,AKPW83,All83,AK84,HK84,CCK87, Cal87]. More recently, the *PTRAN* project under Allen at the IBM T. J. Watson Research Center has begun building a parallelizing compiler capable of generating code for a large class of multiprocessors [BC86,CF87,ABC+87].

The vectorization techniques developed over the last fifteen years are now found in virtually every compiler for a vector machine. Parallelization for multiprocessors is not as advanced. To the best of our knowledge, only one commercial multiprocessor, the Alliant FX, currently has a parallelizing compiler [All85].

For many years, compaction techniques were of interest primarily in the microprogramming community [RT74,DT76,TG76,Woo78,TTTY77,Mal78,LDS80]. Early techniques were quite limited; the focus of attention was on developing good algorithms for scheduling basic blocks. An important breakthrough was achieved by Fisher. He defined *trace scheduling*, the first general compaction technique that applied across basic block boundaries [Fis79,Fis81]. Since then, Fisher and others have explored improvements to and generalizations of trace scheduling [TTT81,FLS81,LA82,SDJ84,Ell85,Nic85b]. Trace scheduling is the centerpiece of Multiflow's compiler for its Trace series of scientific computers [Mul87].

Parallelization has had its detractors. In 1967, Amdahl observed that when a program is parallelized, the execution time is dominated by the portion that is not speeded up. If a fraction $1/n$ of the execution time of a program cannot be parallelized, then parallelization can achieve at most a factor of n speedup over a sequential machine. This is known as Amdahl's Law [Amd67]. Amdahl asserted that in real programs parallelization would have no effect on at least 10% of the execution time, so the cost-effectiveness of building a parallel machine was worse than building a sequential machine that ran ten times faster. This argument was supported by early experiments which suggested little parallelism could be extracted from programs in practice [TF70,RF72].

Researchers in the parallelization community have argued tirelessly against Amdahl's Law.[4] However, the law is certainly correct, and the conclusion that a faster sequential machine is more cost-effective was also correct in 1967. Am-

[4]Virtually every treatise on parallelization of any length—including this one—devotes a few paragraphs to Amdahl's Law.

dahl's Law shows that parallelization has little effect unless the entire program can be parallelized. For scientific programs, later experiments showed that there is substantial parallelism available in scientific programs using only dependency information [NF84]; for this class of programs, the fraction of code that cannot be parallelized is sufficiently small that Amdahl's Law does not present a significant barrier.[5] Furthermore, the cost-effectiveness argument is, in the long run, irrelevant: there is an upper limit on the speed of sequential processors; beyond this point, the only hope for achieving higher performance is through parallelism.

1.3 Dissertation Overview

This dissertation is broadly organized into three major sections. The first section defines our primitive program transformations and discusses their theoretical properties. The second section develops algorithms from the primitive transformations. The third section discusses extensions to the basic system, implementation issues, and related work.

The first section consists of Chapters 2 through 4. Chapter 2 develops the model of computation and the measure of program improvement. Chapter 3 defines the primitive transformations. Chapter 4 investigates the theoretical strengths and weaknesses of the transformational system with respect to the model of Chapter 2.

The second section consists of Chapters 5 through 10. Chapter 5 develops a compaction algorithm *compact-global*; this is shown to be a generalization of trace scheduling. Chapter 6 develops *perfect pipelining*, a transformation for taking limits of infinite sequences of the primitive transformations. The theoretical results of Chapter 6 are used to develop two software pipelining algorithms in Chapters 7 and 8. Chapter 7 presents a simple software pipelining algorithm for all loops; this is shown to be a generalization of *doacross* [Cyt84]. Chapter 8

[5]See [Nic84] for a discussion of what was wrong with the early experiments.

develops a software pipelining algorithm that generates optimal code for a subset of all loops. In Chapter 9 we consider nested loops. The results of Chapters 6 and 9 are used in Chapter 10 to develop a general software pipelining algorithm for nested loops; this is shown to be at least as general as the *wavefront method* [Mur71,Lam74,Kuh80,Wol87].

The third section consists of Chapters 11 through 13. Chapter 11 discusses several extensions to the basic framework, such as including transformations that alter the dependencies of a program and handling realistic resource constraints. Chapter 12 discusses related work and open problems in the area of compaction-based parallelization. Chapter 13 gives an overview of the current implementation, summarizes our results, and compares our compaction-based approach with parallelization for vector machines. The details of some technical points are included in the appendix.

1.4 A Primer on Dependency Analysis

We study program transformations based on the *data dependencies* of a program. There is a dependency between two program statements if they can refer to the same memory location and one statement writes that location. Dependencies represent sequential relationships between statements that must be preserved. Any compiler using transformations based on data dependencies must perform *dependency analysis*. In general, dependency analysis is a procedure D which takes two program statements as input and returns *yes* if there is a dependency between the two statements and *no* if there is no dependency. However, determining whether two statements are data dependent is undecidable. Thus, we settle for a *conservative* procedure: if $D(a,b) = no$ there is definitely no dependency, but if $D(a,b) = yes$ there may or may not be a dependency.

Dependency analysis has been a thriving area of research for twenty years. Bernstein first proposed the idea of automatic dependency analysis [Ber66]. Many of the procedures for dependency analysis and the formalization of *dependency graphs* were introduced by Kuck's group. Dependency graphs have become the focus of research in dependency analysis [KMC72,Kuc76,Ban76,Tow76,KKP+81, All83,BC86,FOW87,HPR88a,HPR88b]. The nodes of the dependency graph are program statements. There is a directed edge between statements n and n' if the statements are dependent and n is executed before n'. Control information is usually also included as distinct type of edge. Although interest in dependency analysis has historically been confined to the parallelization community, there are other applications; see, for example, [ASKL81,OO84,HPR88a].

The dependency graph of a program is normally computed once prior to parallelization. This has the advantage of making procedure D very inexpensive: $D(a,b)$ is true if there is an edge (a,b) in the dependency graph. For compaction algorithms such as trace scheduling, the program changes so drastically during parallelization that it is more convenient to compute dependency information dynamically using incremental flow analysis of the program [Nic84].

Recently, dependency analysis has been studied as an abstract interpretation of Pascal-like languages [Jou87]. An algorithm for computing dependency and alias information in higher-order languages with assignment (also based on abstract interpretation) is presented in [NPD87]. These, together with the results of [HPR88b] on the adequacy of dependency graphs, are recent contributions to the area dependency analysis.

For our purposes, there are two types of dependencies: *true* and *false*. There is a true dependency between two statements if one reads a value the other writes. For example, in Figure 1.3, there is a true dependency between the first and second statements. False dependencies are caused by the reuse of storage. There is a false dependency between two statements if they access the same storage location and at least one writes that location, but no value flows from one statement to the other. For example, in Figure 1.3 there are false dependencies

a := b + 1;
c := a;
a := d;

Figure 1.3: A program fragment with true and false dependencies.

between the first and third statements and the second and third statements caused by the reuse of *a*. In standard dependency graph terminology, the false dependency between the first and third statements is an *output* dependency, while the dependency between the first and second statements is an *anti*-dependency [KKLW80].

We sometimes distinguish whether a dependency between statements in a loop is *loop-carried* or *loop-independent* [All83]. A loop-carried dependency is between statements in different iterations of a loop, while a loop-independent dependency is between statements in the same iteration.

Dependency analysis is not the focus of this dissertation. We only require that there is some conservative procedure for deciding whether two statements are dependent. When it is necessary to explicitly discuss dependency analysis, we adopt either the compaction or dependency graph model, depending on which is more convenient. Our results are, of course, relative to the precision of the dependency analysis.

Chapter 2

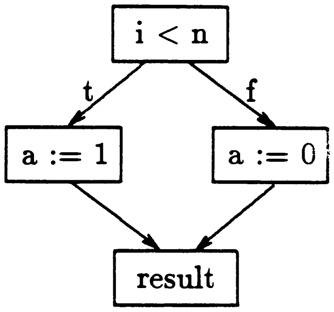
The Model

This chapter provides the formal foundation for our study of parallelizing transformations. There are four sections. The first section informally describes the programming language in which transformations are performed. The second section provides a formal definition of the language and an operational semantics. The third section uses the operational semantics to define a relation $\leq_p$ that measures when one program is "more parallel" than another. The fourth section shows that the definition of $\leq_p$ is too weak to reason about some interesting cases; this leads to a more powerful definition of $\leq_p$.

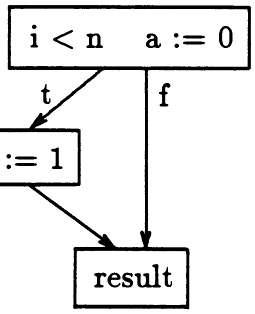
The model of computation and the relation $\leq_p$ are motivated by synchronous parallel computation and compaction-based parallelization. This model is accurate for synchronous parallel computers, such as pipelined and Very Long Instruction Word machines [Fis83,FO84]. In subsequent chapters we show that many of the important techniques of parallelization for synchronous and asynchronous parallel machines can be expressed using our model and transformational system.

2.1 A Simple Language

In this section we give an informal description of SPL, a Simple Parallel Language. SPL is an abstraction of the model of computation used to define *percolation*



(a) A sample program.



(b) A parallelized program.

Figure 2.1: Two versions of the same program.

scheduling [Nic85b]. We have minimized the details of language design while keeping the language rich enough to allow discussion of the important problems. SPL is not so much a "real" programming language as a tool convenient for discussing parallelizing transformations.

SPL is graphical; a program is represented by a control flow graph as in Figure 2.1a. Each node in the graph contains zero or more primitive statements. These statements are divided into two categories: assignments and tests. The evaluation of an assignment updates the store, while tests affect the flow of control. Execution begins at the start node and proceeds sequentially from node to node. When control reaches a particular node, all statements in that node are evaluated concurrently; the assignments update the store and the tests return the next node in the execution sequence (see discussion below). Statements evaluated in parallel perform all reads before any assignment performs a write.

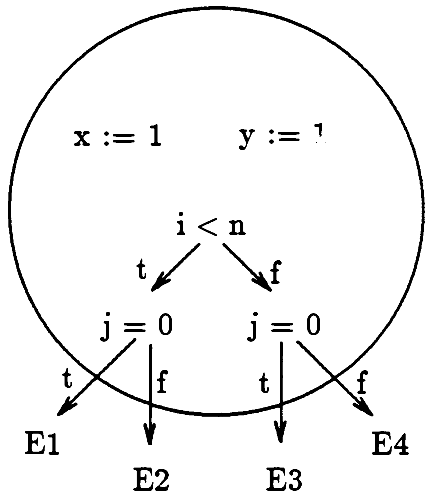


Figure 2.2: A program node.

Write conflicts within a node are not permitted.

Care must be taken to define how multiple tests are evaluated in parallel. The set of tests within a node is given as a directed acyclic graph (*dag*). Each test in the dag has two successors corresponding to its true and false branches. A successor of a test is either another test or a *name*; a name is a pointer to a program node. We require that the dag of tests be *rooted*—that it have a single element with no predecessors. To evaluate a dag in a state, select the (unique) path from the root to a name such that the branches on the path correspond to the value (true or false) of the corresponding test in the state. Evaluation of the dag returns the node name that terminates this path. An example node is given in Figure 2.2. The node contains three tests and two assignments. The tests are organized as a tree; assuming that all of the tests evaluate to true, the next node in the execution sequence is E1.

On a real machine the evaluation of multiple tests can be very sophisticated to exploit parallelism. For example, the tests can be evaluated simultaneously and a fast algorithm used to select the correct node name from the boolean results. A hardware mechanism that efficiently implements general dags of tests is described in [KN85]. Implementing trees of tests is discussed in [Fis80,Ebc87]. Less gen-

$$
\begin{array}{lcl}
\text{Bool} & = & \text{tt} + \text{ff} \\
\text{Loc} & = & \mathcal{Z} \\
\text{Store} & = & \text{Loc} \to \text{Val} \\
\text{Assign} & = & \text{Store} \to \text{Loc} \times \text{Val} \\
\text{Test} & = & \text{Store} \to \text{Bool}
\end{array}
$$

Figure 2.3: Basic domains.

eral multiway jump mechanisms are used in many horizontal microengines and Multiflow's TRACE architecture [Mul87].

SPL is powerful enough to model execution of a tightly-coupled parallel machine at the instruction level. It is at this level that our transformational system extracts parallelism from programs. The differences between our model and a real machine are the absence of resource constraints and the assumption that all statements have unit execution time. Methods for compensating for these assumptions are well-known [FERN84,Ell85]; we address these issues in Chapter 11. A sample SPL program is shown in Figure 2.1a. Note that this program has only one statement per node; such a program is *sequential.* Another, more parallel version of the same program is given in Figure 2.1b.

2.2 Formal Definition of SPL

The formal definition of SPL and its operational semantics provide a framework for proving properties of program transformations. In subsequent sections we develop a formalism for our transformations; this formalism uses the operational semantics of SPL to define when one program is more parallel than another. The operational semantics of SPL closely follows the structural style advocated by Plotkin [Plo].

Figure 2.3 lists the basic domains of SPL. Val is a domain of basic values—integers, floating-point numbers, etc. An assignment, a function of type Assign, deviates from the standard approach in that it does not return an updated store. Instead, an assignment returns a pair $\langle l, v \rangle$, where v is the new value of location l. This allows us to define the parallel execution of several assignments as the parallel binding of the new values to the updated locations (recall that write conflicts are not permitted). A *program* is a tuple $\langle N, n_0, F \rangle$ where:

N is a set of *nodes*

$n_0 \in N$ is the start node

$F \subseteq N$ is the set of final nodes

A node is a pair $\langle A, C \rangle$ where:

A is a set of *assignments*

C is a *dag*; a four-tuple $\langle B, \mathit{select}, r, H \rangle$ where:

B is a set of *tests*

$\mathit{select} : B \times \mathrm{Bool} \rightarrow B + H$ is an edge function

r is the root test or a node name

H is a set of node names

We assume that all occurrences of a statement are uniquely identifiable. In what follows, s and s' range over stores; variants of v, l, a, and t range over values, locations, assignments, and tests respectively. We assume that assignments and tests are total atomic actions of type Assign or Test. We use n for both the name of a node and the node itself; the meaning is always clear from the context.

The transformations we define require knowledge of the locations that are read and written by statements to model *dependency analysis*. Dependency analysis determines when two program statements may refer to the same memory location (see Section 1.4). The analysis is used to determine when it is safe to execute statements in parallel. We define $\mathit{write}(a, s)$ to be the location written by assignment a in store s; $\mathit{read}(a, s)$ is the set of locations read by assignment (or test) a in store s.

Let $n = \langle A, C \rangle$, $C = \langle B, select, r, H \rangle$, and $A = \{a_i\}$

Constraints on C:

The directed graph induced on B and H by *select* is:

connected

acyclic

has a unique root r.

Constraint on A:

$\forall s, i, j$ where $i \neq j$, $write(a_i, s) \cap write(a_j, s) = \emptyset$

Constraints on n:

$\forall n \in N\ n_0 \notin succ(n)$

$\forall n \in F\ succ(n) = \emptyset$

$\forall n \in F$ where $n = \langle A, \langle B, select, r, H \rangle \rangle$, $A = \{result\} \wedge B = \emptyset$

Figure 2.4: Program constraints.

In Section 2.1, we discussed well-formedness conditions and semantic constraints not captured by the above description. These requirements are listed in Figure 2.4. The constraints ensure that the dag of tests is well-formed and that two assignments in a node cannot write the same location. In addition, the start node should have no predecessors and a final node should have no successors. A final node contains a distinguished statement, *result*, that reads and returns the result of the computation. We assume that *result* returns the entire final store, although any set of store locations would do.

2.2.1 Useful Functions on Program Nodes

Figure 2.5 lists several useful functions on program nodes. We briefly describe each function. *Succ* returns the immediate successors of a node; when it is convenient we refer to an edge (m, n) instead of writing $n \in succ(m)$. *Pred* returns the immediate predecessors of a node. The function *op* returns the statements in a node. *Node*(x) is the node containing statement x (we assume there is a way of distinguishing between multiple copies of the same statement). The function *attach* builds a new dag out of two dags and a test x. The second and third arguments correspond to the true and false branches of x respectively; x is the root of the resulting dag. *Reach* returns the subdag reachable (in the sense of a directed graph) from a test or name t. The function *replace-edge* returns a dag with names pointing to node n replaced by names pointing to n'.

2.2.2 An Operational Semantics of SPL

An *operational semantics* for a language describes the execution of programs written in that language. Figure 2.6 gives an operational semantics for SPL. The semantics consists of a set of rewriting rules in the style of inference rules of formal logic. There are two types of transitions: $\rightsquigarrow$, which defines transitions within a node, and $\rightarrow$, which defines transitions between nodes. Each transition between nodes corresponds to one step of a synchronous parallel machine. Rules are read as stating that the assertion below the line holds if the assertions above the line hold.

The first two rules deal with evaluation of a dag of tests. Consider the first rule. In configuration $\langle C, s, t\rangle$, if t is a test, then the next configuration is $\langle C, s, select(t, \mathrm{tt})\rangle$ if t evaluates to true in store s; otherwise, the next configuration is $\langle C, s, select(t, \mathrm{ff})\rangle$. The second rule returns the node name that terminates the path selected through the dag of tests—the configuration $\langle C, s, n'\rangle$ rewrites to n' if n' is a node name.

The third rule describes the parallel evaluation of assignments. Each assignment a_i is evaluated in store s. The resulting pairs $\langle l_i, v_i\rangle$ represent locations l_i to be written with new values v_i. (Recall that two assignments cannot write

succ: Node $\rightarrow \mathcal{P}$(Node)
$succ(n) = H$ where $n = \langle A, \langle B, select, r, H\rangle\rangle$

pred: Node $\rightarrow \mathcal{P}$(Node)
$pred(n) = \{n' | n \in succ(n')\}$

op: Node $\rightarrow \mathcal{P}$(Assign + Test)
$op(n) = A \oplus B$ where $n = \langle A, \langle B, select, r, H\rangle\rangle$

node: Assign + Test $\rightarrow$ Node
$node(x) = n$ where $x \in op(n)$

attach: Test $\times$ Dag $\times$ Dag $\rightarrow$ Dag
Let $C_1 = \langle B_1, sel_1, r_1, H_1\rangle$, $C_2 = \langle B_2, sel_2, r_2, H_2\rangle$
$attach(t, C_1, C_2) = \langle B_1 \cup B_2 \cup \{t\}, sel[\langle t, \mathrm{tt}\rangle \leftarrow r_1, \langle t, \mathrm{ff}\rangle \leftarrow r_2], t, H_1 \cup H_2\rangle$
where $sel = sel_1 \sqcup sel_2$

reach: Dag $\times$ (Test + Name) $\rightarrow$ Dag
Let $C = \langle B, sel, r, H\rangle$

$$reach(C, t) = \begin{cases} \perp & \text{if } t \notin B, H \\ \langle \emptyset, \lambda x.\ \perp, t, \{t\}\rangle & \text{if } t \in H \\ attach(t, reach(C, sel(t, \mathrm{tt})), reach(C, sel(t, \mathrm{ff}))) & \text{otherwise} \end{cases}$$

replace-edge: Dag $\times$ *list-of*(Name) $\rightarrow$ Dag
$replace\text{-}edge(\langle B, select, r, H\rangle, n_1, n_1', \ldots, n_i, n_i') = \langle B, select', r', H'\rangle$
where $H' = H\left[n_j'/n_j\right]$
$select' = select\left[\langle z, b\rangle \leftarrow n_j'\right]$
where $select(z, b) = n_j$

$$r' = \begin{cases} n_j' & \text{if } r = n_j \\ r & \text{otherwise} \end{cases}$$

Figure 2.5: Functions on program nodes.

$$\frac{C = \langle B, select, r, H\rangle,\ t \in B,\ select(t, t(s)) = t'}{\langle C, s, t\rangle \rightsquigarrow \langle C, s, t'\rangle}$$

$$\frac{C = \langle B, select, r, H\rangle,\ n' \in H}{\langle C, s, n'\rangle \rightsquigarrow n'}$$

$$\frac{A = \{a_i\}, a_i(s) = \langle l_i, v_i\rangle,\ s[\ldots, l_i \leftarrow v_i, \ldots] = s'}{\langle A, s\rangle \rightsquigarrow s'}$$

$$\frac{n = \langle A, C\rangle, C = \langle B, select, r, H\rangle,\ n \notin F,\ \langle C, s, r\rangle \overset{*}{\rightsquigarrow} n',\ \langle A, s\rangle \rightsquigarrow s'}{\langle n, s\rangle \rightarrow \langle n', s'\rangle}$$

Figure 2.6: The operational semantics of SPL.

the same location in the same step—thus, the l_i are distinct). The configuration $\langle A, s\rangle$ rewrites to s', where is s' is s with the new values written in the appropriate locations.

The first three rules describe the internal ($\rightsquigarrow$) transitions of a node. The final rule describes transitions between nodes. Let $\langle n, s\rangle$ be a configuration where n is not a final node. If the evaluation of the dag of tests produces a node n' and the parallel evaluation of the assignments produces a store s', then the next configuration is $\langle n', s'\rangle$.

A rewriting sequence is an execution history of the computation of a program. For our purposes, a complete rewriting sequence contains much irrelevant detail; in particular, we are rarely interested in the internal evaluation of a node (the $\rightsquigarrow$ transitions). The following definition puts a rewriting sequence at the right level of abstraction for viewing execution as transitions from nodes to nodes:

Definition 2.2.1 (Execution Trace) The *execution trace* of program P in initial store s, written $T(P, s)$, is the sequence $\langle n_0, s_0\rangle \rightarrow \langle n_1, s_1\rangle \rightarrow \langle n_2, s_2\rangle \rightarrow \ldots \rightarrow \langle n_k, s_k\rangle$ where $s_0 = s$, n_0 is the start node of P, and $n_k \in F$. Traces are defined only for terminating computations.[1]

Finally, we sometimes need to refer to statements that are actually involved in the internal ($\leadsto$) transitions of a node. To allow us to do this concisely, we define the set of statements executed by a node.

Definition 2.2.2 Let $exec(n, s)$ be the set of statements executed by n in store s. If $n = \langle A, C\rangle$ where $C = \langle B, sel, r, H\rangle$, then

- if $x \in A$ then $x \in exec(n, s)$
- if $r \in B$ then $r \in exec(n, s)$
- if $\langle C, s, r\rangle \overset{*}{\leadsto} \langle C, s, t\rangle$ and $t \in B$ then $t \in exec(n, s)$

2.3 A Formalization of Parallelism

In this section we develop a formal account of our transformations. This allows us to make precise claims about our transformations and to make comparisons with other transformations. We are interested in transformations that exploit only flow of control and data dependency information; this is a natural and large class of transformations (including our transformations) dominating the literature on parallelization. Examples of transformations in this class include: vectorization [KKLW80,Wol82,AK82], the wavefront method [Mur71] (or hyper-plane method [Lam74]), loop distribution [Kuc76], loop interchange [Wol82,AK84,Wol86], trace scheduling [Fis79,Fis81], and doacross [Cyt86].

Our goal is to define a natural relation, independent of any specific transformations, that captures the usual notion of program optimization found in the literature on parallelization. This notion consists of two parts: first, transformations reorder program statements based on dependency information (thus preserving program semantics), and second, a transformed program is better

[1]While the results should be extensible to infinite computations, this adds complexity to the formalism and is not important to the issues we address.

(faster) than the original program. We introduce a preorder on programs, "sim" (for similarity), that captures when one program approximates the control and dependency structure of another. We then introduce a relation $\leq_p$ that is a restriction of **sim**. If $P \leq_p P'$, then statements in P' are executed at least as early as corresponding statements in P—in this sense, P' is more parallel than P.

Informally, a program P is **sim** to P' if P' executes the same statements as P in an order compatible with the data and control dependencies present in P. P' may, however, have additional statements on some paths that do not affect the output of the program. The program in Figure 2.1b has more statements on some paths than the program in Figure 2.1a, but the two programs are semantically equivalent. A transformation respecting **sim** can only take advantage of control and dependency information computable from a program; other semantic properties of a program are not exploited.

(There are parallelization techniques which make good use of other semantic properties of programs. For example, tree height reduction [KM75] exploits the associativity of arithmetic operations to balance the parse tree of an arithmetic expression. The expression can then be evaluated on a parallel machine in time proportional to the depth of the transformed parse tree. We incorporate these transformations into our framework in Chapter 11.)

The purpose of **sim** is to establish a sensible correspondence between statements in traces of P and statements in traces of P'. It is not enough to simply ensure that copies of statements from P appear in P', since there is no guarantee that dependencies are preserved.

Definition 2.3.1 We say that y *depends on* x in trace $T(P, s)$, written $x \prec y$, if y reads a value written by x. Formally, let $\langle n_0, s_0 \rangle \xrightarrow{*} \langle n_i, s_i \rangle \xrightarrow{*} \langle n_j, s_j \rangle$. Then $x \prec y$ if $x \in \mathit{exec}(n_i)$, $y \in \mathit{exec}(n_j)$, $\mathit{write}(x, s_i) \subseteq \mathit{read}(y, s_j)$, and there is no statement z in n_k for $i < k < j$ such that $\mathit{write}(x, s_i) = \mathit{write}(z, s_k)$.

The relation $\prec$ models *true* dependencies [KKLW80], which correspond to actual definitions and uses of values during execution. This is not conservative dependency analysis—the relation $\prec$ precisely captures the flow of values through an execution of a program. This is all that is required to define the relation **sim**.

Definition 2.3.2 (Similarity) P **sim** P' if and only if there exists a function f satisfying:

$$\forall s \quad x \prec y \text{ in } T(P,s) \Rightarrow f(x) \prec f(y) \text{ in } T(P',s) \wedge$$
$$f(x) \prec y' \text{ in } T(P',s) \Rightarrow x \prec f^{-1}(y') \text{ in } T(P,s)$$

where f is one to one from statements in $T(P,s)$ to statements in $T(P',s)$ and $f(x)$ is an occurrence of x.

The function f provides a mapping demonstrating that P' preserves the dependency structure of P. To prove that **sim** is a sensible relation, we show that it is sound with respect to the operational semantics.

Definition 2.3.3 The result $R(P,s)$ of a computation is the value returned by the statement *result*; this is the last store of $T(P,s)$. Two programs P and P' are *strongly equivalent* if $\forall s\ R(P,s) = s' \Leftrightarrow R(P',s) = s'$.

Two strongly equivalent programs either both compute the same result or both diverge in all possible computations. The relation **sim** imposes a much stronger condition: not only must the programs compute the same values, they must compute in "similar" ways. Because we are interested in program transformations that rely on dependency information, our transformations respect **sim**. In Chapter 11 we consider additional transformations that alter the dependency structure of a program (and therefore do not respect **sim**). We prove that if P **sim** P', then P and P' are strongly equivalent. The following development provides the machinery needed to prove this theorem.

Definition 2.3.4 Consider a trace $T(P,s)$. The *trace dependency graph* is $D = (V,E)$, where V is the set of all statements in the trace and there is an edge between every pair of dependent statements; i.e., $E = \{(x,y) | x \prec y\}$. Multiple copies of the same statement appearing in the trace are distinct nodes in the trace dependency graph.

The following definition describes the dependency graph of a single statement. (This is related to the notion of a slice in static analysis of programs [Wei84].) Lemma 2.3.6 uses this definition to show that corresponding statements (under **sim**) have isomorphic dependency graphs. Note that statements with isomorphic dependency graphs compute the same value.

Definition 2.3.5 Given $D = (V, E)$, the trace dependency graph of $T(P, s)$, let $D_x = (V_x, E_x)$, the restriction of D to a statement x, be the subgraph induced by all statements on which x directly or indirectly depends. Defined inductively:

- $x \in V_x$
- $x \in V_x \wedge (y, x) \in E \Rightarrow y \in V_x \wedge (y, x) \in E_x$

The following lemma shows that any choice of f in Definition 2.3.2 preserves the dependency structure of P and P'.

Lemma 2.3.6 Let P **sim** P'. Let f be any mapping demonstrating that P **sim** P'. Consider any pair of traces $T(P, s)$ and $T(P', s)$. Let D_x and $D'_{f(x)}$ be the dependency graphs of x in $T(P, s)$ and $f(x)$ in $T(P', s)$ respectively. Then $D_x \cong D'_{f(x)}$.

Proof: The proof is by induction on the depth of the graph D_x.

Basis: The depth of D_x is one. Then x depends on nothing but the initial store s. This implies that $f(x)$ depends on nothing but the initial store, since $f(x) \prec y'$ in $T(P', s) \Rightarrow x \prec f^{-1}(y')$ in $T(P, s)$. So $D_x \cong D'_{f(x)}$.

Step: Assume that $D_y \cong D'_{f(y)}$ for all statements with dependency graphs of depth $i - 1$ or less. Let D_x have depth i. Then x depends on a set of statements $\{y_1, \ldots, y_j\}$ with dependencies graphs of depth $i - 1$. By the hypothesis,

$D_{y_k} \cong D'_{f(y_k)}$. Furthermore, for all j, $f(x) \prec f(y_j)$, and $f(x)$ depends on no other statements because $f(x) \prec y'$ in $T(P', s) \Rightarrow x \prec f^{-1}(y')$ in $T(P, s)$. So $D_x \cong D'_{f(x)}$. □

Theorem 2.3.7 (**Soundness of sim**) If P **sim** P', then P is strongly equivalent to P'.

Proof: Let f be the dependency-preserving mapping used to show P **sim** P'. Consider any pair of traces $T(P, s)$ and $T(P', s)$. The statement *result* is distinguished and occurs only once in each trace; therefore, f maps *result* in $T(P, s)$ to *result* in $T(P', s)$. By Lemma 2.3.6, $D_{result} \cong D'_{result}$. The statement *result* returns the final store, so the final store of the two traces is the same. Because the choice of traces was arbitrary, we conclude that the two programs always have the same final store. Therefore, $\forall s\ R(P, s) = s' \Leftrightarrow R(P', s) = s'$. □

We now introduce the relation $\leq_p$. If $P \leq_p P'$, then all statements in P' are executed at least as early in the execution trace as corresponding statements in P.

Definition 2.3.8 Let $x \in exec(n_i)$ in $T(P, s)$. The position of x, written $pos(x)$, is i.

$$\begin{aligned} P \leq_p P' \quad &\Leftrightarrow \quad P \textbf{ sim } P' \wedge \forall s\ pos(x) = i \text{ in } T(P, s) \Rightarrow \\ &\qquad pos(f(x)) = j \text{ in } T(P', s) \text{ where } i \geq j \end{aligned}$$

The purpose of $\leq_p$ is to measure improvement: if $P \leq_p P'$ then P' should be a better program than P. Ideally, $\leq_p$ would be a partial order. A partial order is reflexive ($P \leq_p P$), transitive ($P \leq_p P' \wedge P' \leq_p P'' \Rightarrow P \leq_p P''$), and antisymmetric ($P \leq_p P' \wedge P' \leq_p P \Rightarrow P = P'$). It is a straightforward to verify that $\leq_p$ is reflexive and transitive. It is not, however, antisymmetric. A counter-example is given in Figure 2.7. In this example, both programs have the same traces, so $P \leq_p P'$ and $P' \leq_p P$. However, the two programs are different.

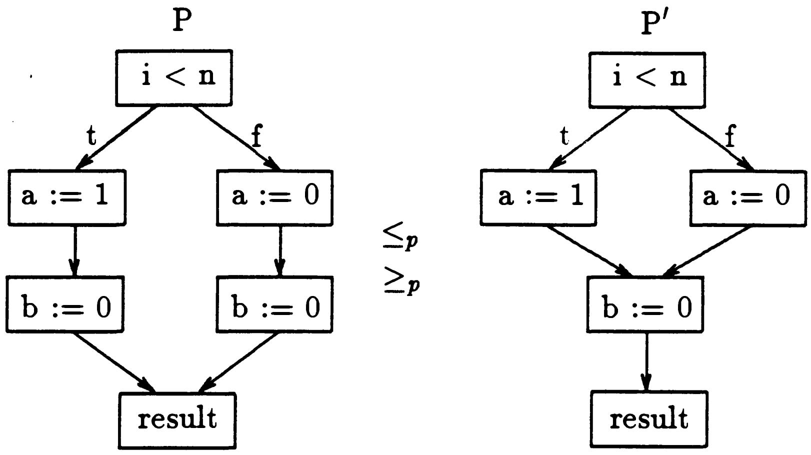


Figure 2.7: Programs where $P \leq_p P'$ and $P' \leq_p P$.

The problem arises because $\leq_p$ is based on behavioral or runtime properties—how the program executes—rather than the syntactic representations of programs. In what sense, then, does $\leq_p$ measure parallelism if two distinct programs can each be better than the other? The following theorem shows that if $P \leq_p P'$ and $P' \leq_p P$, then P and P' are indistinguishable in the operational semantics. In every (finite) computation, at each time step, P and P' execute the same statements—thus neither is "more parallel" than the other.

Theorem 2.3.9 If $P \leq_p P'$ and $P' \leq_p P$, then P and P' are *behaviorly equivalent*; that is,

$$\forall T(P,s), T(P',s)\ exec(n_i, s_i) = exec(n'_i, s_i)$$

where $\langle n_i, s_i \rangle$ is the ith step of $T(P,s)$ and $\langle n'_i, s_i \rangle$ is the ith step of $T(P',s)$.

Proof: Assume $P \leq_p P' \wedge P' \leq_p P$. Let $T(P,s) = \langle n_0, s_0 \rangle \rightarrow \langle n_1, s_1 \rangle \rightarrow \ldots$ and let $T(P',s) = \langle n'_0, s_0 \rangle \rightarrow \langle n'_1, s'_1 \rangle \rightarrow \ldots$. Let f and g be functions used to show $P \leq_p P'$ and $P' \leq_p P$ respectively. The proof is by induction on the length of $T(P,s)$.

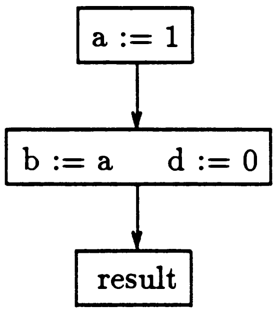
Basis: Consider n_0 and n'_0. Let x be a statement in $exec(n_0)$. From the definition of $\leq_p$, $pos(f(x)) \leq 0$, so $pos(f(x)) = 0$. A symmetric argument shows that statements in $exec(n'_0)$ must be mapped to statements in $exec(n_0)$ under g. Because there is a one to one mapping from statements in n_0 to statements in n'_0 and vice versa (and therefore an onto mapping), we conclude that $exec(n_0) = exec(n'_0)$. This implies that $s_1 = s'_1$.

Step: Assume, for all $j \leq i$, that $s_j = s'_j$. Assume further that f is one-to-one and onto from statements in the first $i-1$ steps of $T(P,s)$ to statements in the first $i-1$ steps of $T(P',s)$, and that g is one-to-one and onto from statements in the first $i-1$ steps of $T(P',s)$ to statements in the first $i-1$ steps of $T(P,s)$.

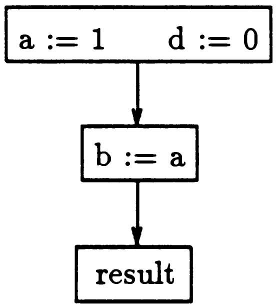
Consider the ith step of $T(P,s)$. Let $x \in exec(n_i)$. By definition of $\leq_p$, $pos(x) \geq pos(f(x))$. But, by the hypothesis, f is one-to-one and onto in the first $i-1$ steps of $T(P,s)$ and $T(P',s)$. Therefore, $pos(x) = pos(f(x))$, which shows that $f(x) \in exec(n'_i)$. A symmetric argument shows that statements in $exec(n'_i)$ must be mapped to statements in $exec(n_i)$ under g. As in the basis step, this implies that $exec(n_i) = exec(n'_i)$ and that $s_{i+1} = s'_{i+1}$. □

Let $\equiv_p$ be the equivalence relation induced by $\leq_p$ (i.e., $P \equiv_p P' \Leftrightarrow P \leq_p P' \wedge P' \leq_p P$). Theorem 2.3.9 justifies $\leq_p$ as a measure of parallelism by showing that programs indistinguishable in $\leq_p$ are indistinguishable in their runtime parallelism. In addition, if $P \leq_p P'$ and $P \not\equiv_p P$, then P' is strictly better than P in the sense that some statement is executed earlier in some trace of P' than in the corresponding trace of P. We write $P <_p P'$ if $P \leq_p P'$ and $P \not\equiv_p P'$. Note that $P <_p P'$ does not imply P' actually has a shorter running time than P—it simply shows that a statement is executed earlier. For example, program P' in Figure 2.8a executes no faster than program P in Figure 2.8b, but $P <_p P'$. The relation $\leq_p$ is motivated by the compaction paradigm—transformations move statements earlier in the schedule, although that alone does not guarantee an improvement in running time. It is certainly the case, however, that scheduling statements earlier cannot lengthen execution time.

Lemma 2.3.10 Let $|T(P,s)|$ be the number of steps in $T(P,s)$. If $P \leq_p P'$, then



(a) A program P.



(b) A program P'.

Figure 2.8: $P <_p P'$ but the execution times are the same.

$|T(P, s)| \geq |T(P', s)|$.

Proof: Trivial. □

2.3.1 An Extension

As developed so far, the relation $\leq_p$ is sufficiently powerful to prove most of the results in this dissertation. However, for some results (particularly those of Chapter 5) $\leq_p$ is too weak. The insistence that every statement appearing in a trace $T(P, s)$ be present in a corresponding trace $T(P', s)$ is too strong. An example is given in Figure 2.9. The relation $\leq_p$ requires a mapping that is one-to-one from statements in traces of P to statements in traces of P'. In this example, P has a trace with two occurrences of x while the corresponding trace of P' has only one occurrence of x. Therefore, even though x in P' occurs at least as early as x in P in every corresponding pair of traces, the relationship $P \leq_p P'$ does not hold.

The problem is that not all the copies of x in P are useful in every computation. The first copy of x is redundant in any computation that also executes the second copy of x. (This is the notion of partially redundant computations of [MR79].) We redefine **sim** to consider only the useful statements appearing in $T(P, s)$ and $T(P', s)$. Intuitively, an assignment is useful if it contributes to the output of the program (*result*) or to a useful test.

A program with a partially redundant test is shown in Figure 2.10. Determining when a test is useful is subtler than determining when an assignment is useful. Intuitively, a test is useful if it can distinguish between two distinct computations. This is not quite right, because **sim** compares the dependency structure of two programs and not just what the programs compute. A more accurate statement is that a test is useful if it can distinguish between two distinct computations with different dependencies. For example, in Figure 2.10, the test $x = 0$ is redundant in any store where $y = 0$. A test t is useless if $\neg t$

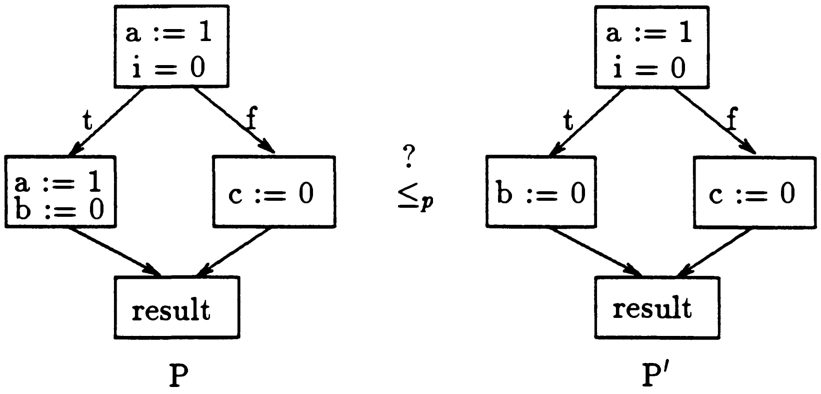


Figure 2.9: An example where $\leq_p$ is too weak.

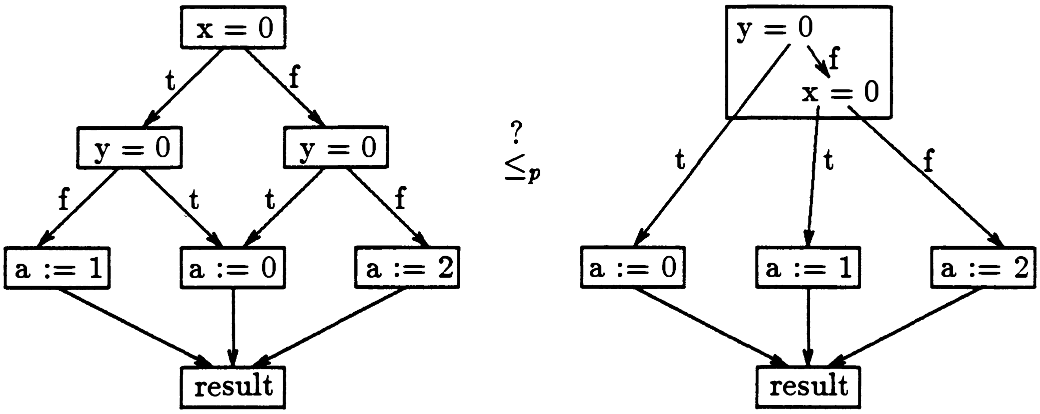


Figure 2.10: The test x = 0 in P is useless in some states.

can be substituted for t and the resulting execution trace has the same useful assignments.

Definition 2.3.11 Let t be a test in the ith step $\langle n_i, s_i \rangle$ of a trace T. We define traces T_t and $T_{\neg t}$ as follows:

$$\begin{aligned} T_t &= T \\ T_{\neg t} &= \langle n_0, s \rangle \xrightarrow{*} \langle n_{i-1}, s_{i-1} \rangle \rightarrow \langle n'_i, s_i \rangle \rightarrow \langle n'_{i+1}, s'_{i+1} \rangle \rightarrow \cdots \\ &\quad \text{where } n'_i \text{ is } n_i \text{ with } \neg t \text{ replacing } t \end{aligned}$$

Definition 2.3.12 (Useful Statements) A statement x in trace T is useful if:

- x is *result*
- $x \prec y$ and y is useful
- x is a test and $T_{\neg x}$ is undefined or $D_{result} \not\cong D'_{result}$ where D_{result} and D'_{result} are the dependency graphs of *result* in T_x and $T_{\neg x}$ respectively.

We redefine **sim** to consider only useful statements in an execution trace.

Definition 2.3.13 (Extended Similarity) P **sim** P' if and only if for all s, $T(P, s)$ and $T(P', s)$ are undefined or

$$\exists f \text{ s.t. } a \prec b \text{ in } T(P, s) \Rightarrow f(a) \prec f(b) \text{ in } T(P', s)$$

where a and b are useful, f is one-to-one and onto from useful statements of $T(P, s)$ to useful statements of $T(P', s)$, and for all a, $f(a)$ is an occurrence of a.

Although the new definition of **sim** does not greatly alter the results of this chapter, the new definition is much stronger than the original definition. Whereas **sim** was a only preorder, it is now an equivalence relation.

Lemma 2.3.14 **sim** is an equivalence relation: P **sim** $P' \Rightarrow P'$ **sim** P.

Proof: The dependency preserving mappings f in the definition of **sim** are one-to-one and onto. Therefore, the inverse mappings f^{-1} show P' **sim** P. □

We redefine $\leq_p$ to compare only useful statements.

Definition 2.3.15

$$P \leq_p P' \Leftrightarrow P \textbf{ sim } P' \wedge \forall s \; pos(x) = i \text{ in } T(P, s) \Rightarrow$$
$$pos(f(x)) = j \text{ in } T(P', s) \text{ where } i \geq j$$

where x is useful and f is a mapping demonstrating that P **sim** P'.

Theorem 2.3.7 and Lemma 2.3.10 hold unchanged for the new definitions of **sim** and $\leq_p$. Theorem 2.3.9 is no longer true. An example is given in Figure 2.11.

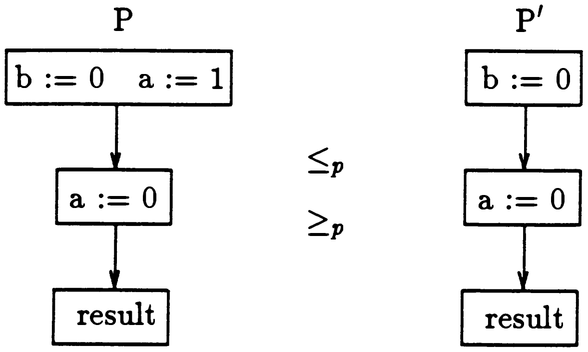


Figure 2.11: $P \leq_p P'$ and $P' \leq_p P$.

In the figure, $P \leq_p P'$ and $P \leq_p P'$, but the two programs are not behaviorally equivalent because P executes a useless statement and P' does not. A slightly weaker version of the theorem does hold. Let *useful-exec*(n, s) be the useful statements n executes in store s.

Theorem 2.3.16 (Weak Equivalence) If $P \leq_p P'$ and $P' \leq_p P$, then

$$\forall T(P,s)\ \textit{useful-exec}(n_i, s_i) = \textit{useful-exec}(n'_i, s'_i)$$

where $\langle n_i, s_i \rangle$ is the ith step of $T(P,s)$ and $\langle n'_i, s'_i \rangle$ is the ith step of $T(P', s)$.

Proof: Same as proof of Theorem 2.3.9. □

This definition of **sim** does not have the anomalies of the original definition and permits us to prove much stronger results. There is a subtlety, however, which arises from the notion of a useful test. We are interested in transformations that use data dependency information to parallelize programs. This is reflected in our model: data dependency information is given, but statements are otherwise uninterpreted—we do not "look inside" a statement and exploit properties besides data dependencies. However, a test may partially or wholly imply the result of a later test, which is potentially troublesome for a model trying to distinguish useful tests. For example, in Figure 2.12, P is **sim** to P' because the first test in P is useless. It is also plausible to decide that the second test in P is useless and

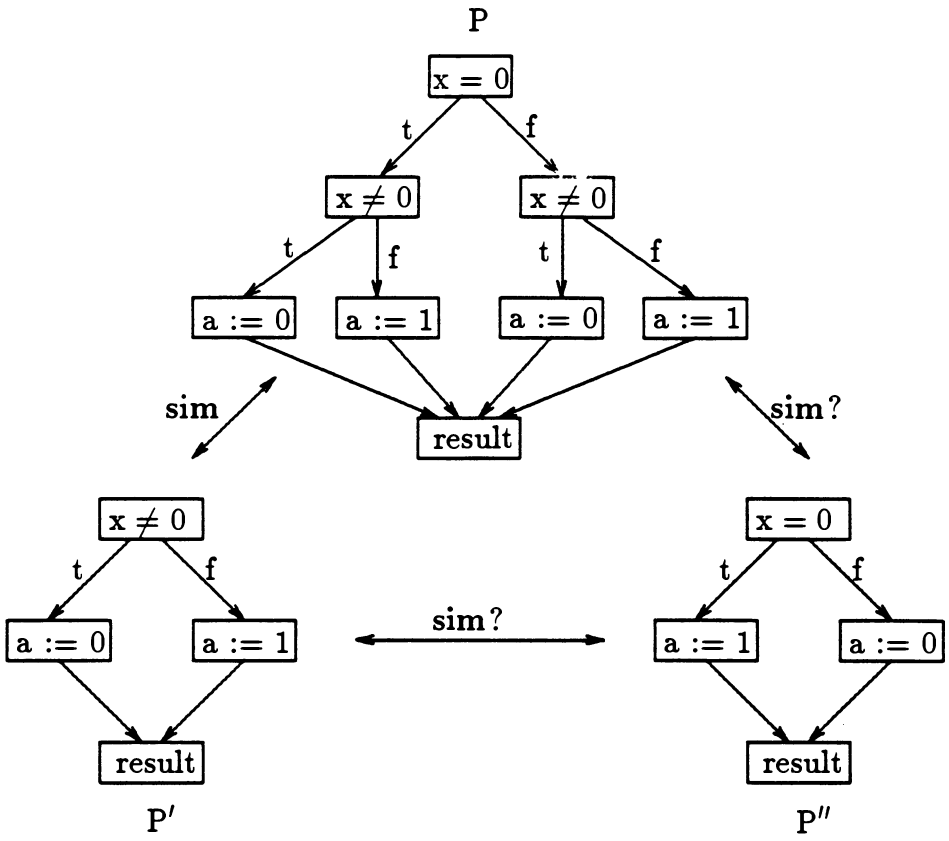


Figure 2.12: A potential difficulty with **sim**.

P is therefore **sim** to P''. But **sim** is an equivalence relation and it cannot be that P' **sim** P''. In our model this is resolved by selecting the last possible test in a trace to be useful. Thus P **sim** P' and not P **sim** P''. This interaction among tests presents no further problems; our transformations exploit only dependency information and cannot themselves create such situations.

2.3.2 Other Relations

The relation $\leq_p$ is not the only possible measure for comparing parallelism in programs. An alternative is to use the ordering induced by some set of program transformations—that is, $P \leq P'$ if P can be transformed into P'. It is fairly easy to get a partial order in this way. In fact, the set of primitive transformations we use induces a partial order on programs. We rejected this approach because nothing can be said about transformations not directly expressible with the chosen primitives. In Chapter 6, we introduce transformations that cannot be implemented using our primitives; $\leq_p$ is crucial to proving properties of these transformations. With $\leq_p$, there is an independent formalism in which transformations can be compared.

Another approach is to find a relation like $\leq_p$ that is a partial order, thereby avoiding the situation of Figure 2.7. A simple way to do this is to define another relation $\leq$ where $P \leq P' \Leftrightarrow P \leq_p P' \wedge (P \not\equiv_p P' \vee P = P')$, thereby making programs that are $\equiv_p$ but not $=$ incomparable. This is too strong, however, because there are useful transformations within the equivalence classes of $\equiv_p$. For example, loop unrolling is an important transformation in many parallelization systems, but it does not change the runtime behavior of a program and is therefore not captured by $\leq$. If there is a natural partial order as powerful as $\leq_p$, then all facets of runtime behavior must be syntactically distinguishable. Since this is not true even for languages with very simple syntax (e.g., the lambda calculus), we feel that there is probably not a natural relation that is also a partial order.

Finally, we use $\leq_p$ to reason about parallelization, but other transformations that preserve dependencies are captured by $\leq_p$. For example, *dead code removal* is a standard compiler optimization for sequential machines that deletes useless statements from a program [ASU86]. Thus, if P' the program obtained by deleting dead code from P, then $P \leq_p P'$. One avenue of future research is to develop a theory of all dependency-based compiler optimizations using $\leq_p$.

Chapter 3

The Core Transformations

The core transformations are the building blocks of our transformational system. These primitive transformations are local, involving only adjacent nodes of the program graph. Though simple, the core transformations can be used to express very powerful code motions. The transformations presented here are based upon the core transformations of percolation scheduling [Nic85b].

This chapter is divided into three major sections. The first section informally describes and illustrates each transformation. The second section sketches a proof of correctness and discusses implementation. The final section provides formal definitions of the transformations and proofs of correctness. This last section is included to provide a complete description of the transformations. As such, the reader may wish to use it only for reference.

3.1 An Overview of the Transformations

There are four primitive transformations: *delete*, *move-op*, *move-test*, and *unroll.* Code motions are accomplished with move-op (for moving assignments) and move-test (for moving tests). Delete improves a program's running time by removing empty nodes. Unroll (loop unrolling) aids parallelization by increasing the number of statements available to other transformations.

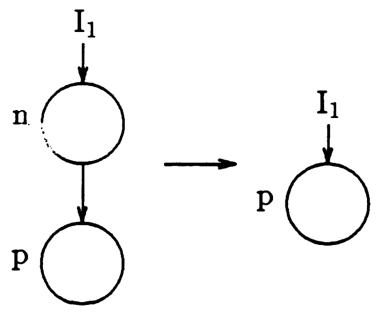


Figure 3.1: The delete transformation.

An illustration is given for each transformation. Only the relevant portion of the program graph is shown; incoming edges are denoted by I_j and exiting edges by E_j.

3.1.1 Delete

The *delete* transformation removes a node from the program graph if it is empty (contains no statements) or unreachable. A node may become empty or unreachable as a result of other transformations. In Figure 3.1, assume n is an empty node. When n is removed from the program graph, n's predecessors are changed to point to n's successor. Note that an empty node has exactly one successor.

3.1.2 Move-op

The *move-op* transformation moves a single copy x of identical assignments from a set of nodes $\{n_j\}$ to a common predecessor node m.[1] This is done if no dependency exists between x and the statements of m and x does not write any storage location live at m. (A storage location is live at m if it could possibly be read in some continuation of a computation from m [ASU86].)

Care must be taken not affect the computation of paths passing through an n_j but not through m. To ensure this, the original node n_j is preserved on the other paths. An illustration of move-op applied to a single assignment x is given

[1]This transformation combines the primitives move-op and unify of [Nic85b].

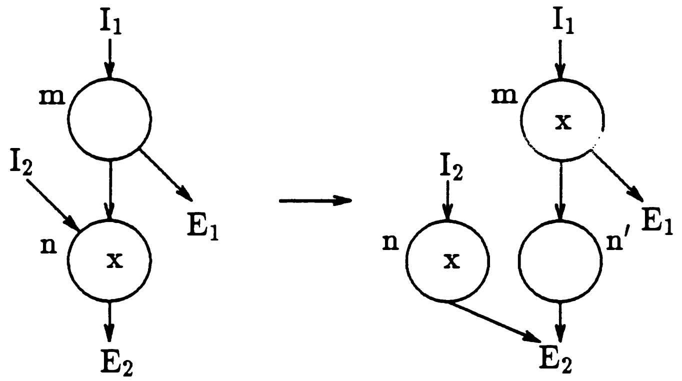


Figure 3.2: The move-op transformation applied to one assignment.

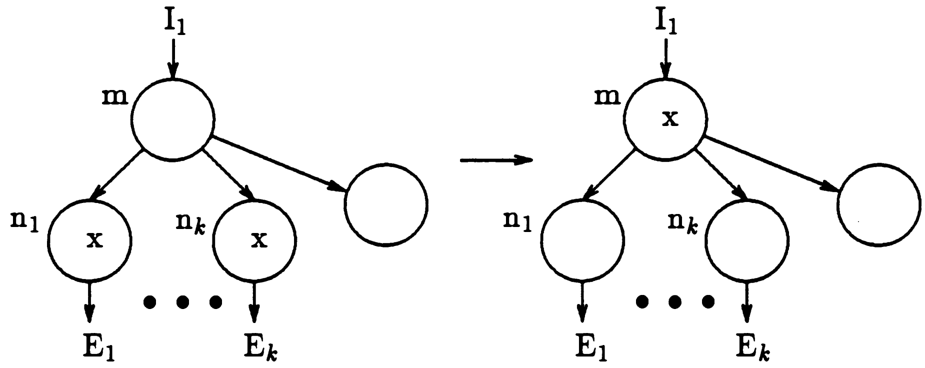


Figure 3.3: The move-op transformation applied to several assignments.

in Figure 3.2; an illustration of move-op applied to several assignments is given in Figure 3.3.

While move-op is defined to move any subset of copies of an assignment that can legally move from a set of nodes to a common predecessor, we assume that move-op always moves all copies. Move-op can also be defined to allow distinct statements that are mutually dependent to move into a node m [Nic85b]; for simplicity, we do not consider this possibility.

3.1.3 Move-test

The *move-test* transformation moves a single copy of a set of identical tests $\{x_j\}$ from a node n to a node m through an edge (m, n) provided that no dependency

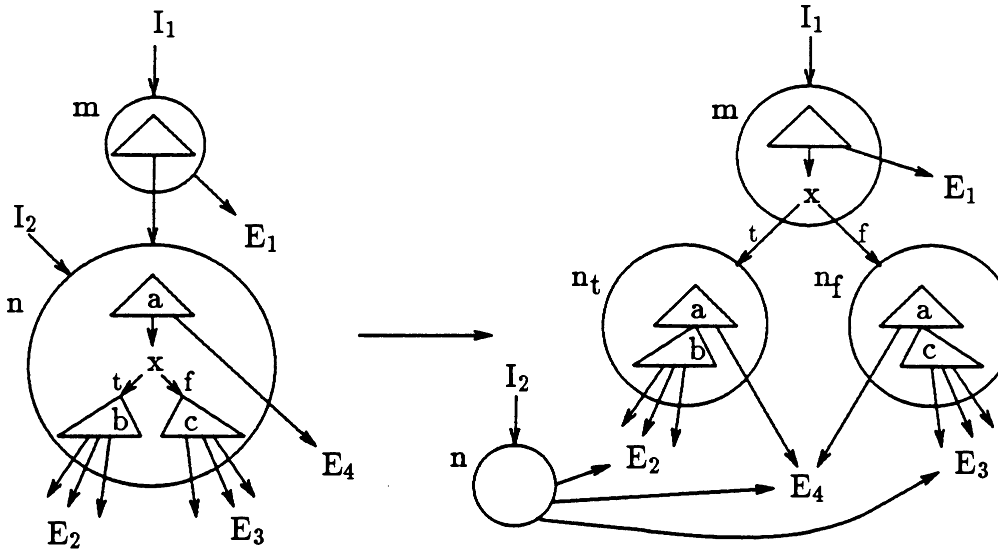


Figure 3.4: The move-test transformation.

exists between x and the statements of m. Paths passing through n but not through m must not be affected; as for move-op, n is preserved on the other paths. Because we allow an arbitrary rooted dag of tests in a node and the tests being moved may come from arbitrary points in that dag, n is split into n_t and n_f, where n_t and n_f correspond to the true and false branches of the x_j. In n_t, the tests on x_j's true branch are inserted in x_j's place; in n_f, the tests on x_j's false branch are inserted in x_j's place. Figure 3.4 illustrates the transformation applied to a single test. In the figure, a represents the dag of tests (in n) not reached by x, b represents the dag of tests reached on x's true branch, and c the dag of tests reached on x's false branch. While move-test is defined to move any subset of copies of a test from a node n to a predecessor m, we assume that move-test always moves all copies.

3.1.4 Unrolling

Loop *unrolling* (or unwinding) is a standard non-local transformation [ASU86]. When a loop is unrolled, the loop body is replicated to create a new loop. Loop unrolling helps exploit fine-grain parallelism by providing a large number of state-

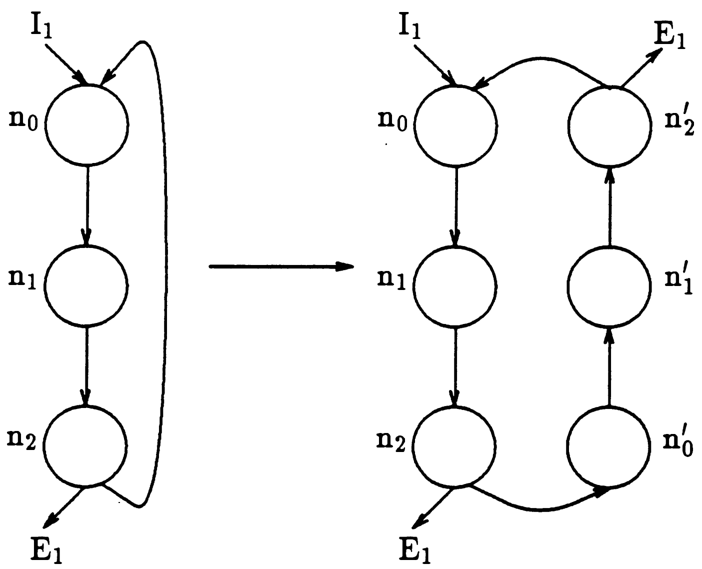


Figure 3.5: The unrolling transformation.

ments (the unrolled loop body) for scheduling. The statements in the unrolled loop body come from previously separate iterations and are thus freer of the order imposed by the original loop. In Chapter 9 we discuss the correct unrolling of multiple nested loops. An illustration of a loop unrolled once is given in Figure 3.5. We use the shorthand u^iL for loop L unrolled i times, where $u^1L = L$. The unrolled iterations of u^iL are $L_0, \ldots, L_{i-1}$.

3.2 Correctness and Implementation

A transformation $\mathcal{T}$ is *correct* if for any program P, $\mathcal{T}(P)$ is strongly equivalent to P. The following theorem justifies the core transformations by showing that they are correct and that some improvement results from their application.

Theorem 3.2.1 Let $\mathcal{T}$ be delete, move-op, or move-test. Then for all P, $P \leq_p \mathcal{T}(P)$. Let $\mathcal{T}$ be unrolling. Then for all P, $P \equiv_p \mathcal{T}(P)$.

Proof: The transformations preserve dependencies and do not remove a statement from any path on which it occurs—thus P **sim** $\mathcal{T}(P)$. For each of delete, move-op, or move-test, if it succeeds, at least one statement appears earlier on at least one path, so $P \leq_p \mathcal{T}(P)$. For unrolling, for any initial state s, $T(P,s) = T(\mathcal{T}(P),s)$ or both traces are undefined, so $P \equiv_p \mathcal{T}(P)$. □

Thousands of applications of delete, move-op, and move-test are required to compact a program, so it is very important that these primitives run quickly. Delete is clearly a constant time operation. Move-op and move-test, however, must check that the moving statement does not write values that are live on other paths. This appears to require global flow analysis. Besides being very slow, this also destroys the local character of the transformations. However, the transformations can be efficiently implemented if flow analysis is maintained incrementally. Global flow analysis is performed once, before compaction begins, and each node in the program graph is attributed with the values live at that node. Flow analysis has been treated extensively in the literature and we do not pursue it further here; see, for example, [All74,All75,AC76,Nie85,ASU86,NPD87]. Move-op and move-test use the live information associated with nodes involved in a transformation to check that the transformation is legal. Furthermore, it is easily verified that a move-op or move-test alters live information only at nodes involved in the transformation. Thus, the entire transformation—the check for legality, the code motion itself, and the update of live information—can be performed locally. Experience with our implementation shows that the check for legality can be performed in constant time in practice; thus, move-op and move-test are both constant time transformations.

3.3 Formal Definitions of the Core Transformations

This section is a reference for the details of the core transformations. Figure 3.6 lists predicates used to define the core transformations. The predicate *empty* is true if its argument is a node with no statements. A node is *unreachable* if it has no predecessors and is not the start node. The predicates *dead*, *no-write-conflict*, and *no-read-conflict* are used to determine when a statement can move from one node to another. These predicates model the dependency analysis of parallelizing compilers. *No-write-conflict*(x, n) is true if the statement x does not read or write a value written by a statement in n. *No-read-conflict*(x, n) is true if x does not write a value read by a statement in n. *Dead(l,n)* is true if location l does not affect further evaluation of the program when control reaches n. Functions used to define the core transformations are presented in Figure 2.5.

Completely formal proofs of correctness of the core transformations require tedious (and relatively uninformative) inductions on the length of the trace of an execution. We present slightly informal proofs that highlight the important properties of the transformations. Minor variations on these proofs show that $P \leq_p \mathcal{T}(P)$ for any core transformation $\mathcal{T}$.

Delete

The formal definition of delete is given in Figure 3.7. A picture is given in Figure 3.1. Note that an empty node has exactly one successor.

Lemma 3.3.1 Delete is correct.

Proof: Let *delete*$(P, n) = P'$. Consider traces $T(P, s)$ and $T(P', s)$. Because delete preserves flow of control and does not remove any statements from P, the trace dependency graphs of the traces are identical. Thus, P is strongly equivalent to P'. □

Move-op

The formal definition of move-op is given in Figure 3.8. A picture is given in Figure 3.2. Note that *no-read-conflict* is used in a dependency test on the statements of n. This is necessary because two statements in a node may read and write

Let $P = \langle N, n_0, F\rangle$ Let $n = \langle A, \langle B, sel, r, H\rangle\rangle$
empty: Node $\rightarrow$ Bool
$empty(n) \equiv A = \emptyset \wedge B = \emptyset$

unreachable: Node $\rightarrow$ Bool
$unreachable(n) \equiv (pred(n) = \emptyset \wedge n \neq n_0)$

no-write-conflict: (Test + Assign) $\times$ Node $\rightarrow$ Bool
$no\text{-}write\text{-}conflict(x, n) \equiv \forall s, a \in A\ (read(x, s) \cup write(x, s)) \cap write(a, s) = \emptyset$
where $write(x, s) = \emptyset$ if x is a test

no-read-conflict: (Test + Assign) $\times$ Node $\rightarrow$ Bool
$no\text{-}read\text{-}conflict(x, n) \equiv \forall s, a \in A\ write(x, s) \cap read(a, s) = \emptyset$
where $write(x, s) = \emptyset$ if x is a test

dead: Loc $\times$ Node $\rightarrow$ Bool
$dead(l, n_i) \equiv \forall s\colon \text{Store}, v\colon \text{Val},\ \langle n_0, s\rangle \xrightarrow{*} \langle n_i, s_i\rangle \rightarrow \langle n_{i+1}, s_{i+1}\rangle \xrightarrow{*} \langle n_k, s_k\rangle \Rightarrow$
$\langle n_i, s_i[l \leftarrow v]\rangle \rightarrow \langle n_{i+1}, s'_{i+1}\rangle \xrightarrow{*} \langle n_k, s_k\rangle$
where $n_k \in F$.

Figure 3.6: Some useful definitions.

delete: Pgm $\times$ Node $\rightarrow$ Pgm

Let $P = \langle N, n_0, F \rangle$

$n = \langle A, C \rangle$

$C = \langle B, select, r, H \rangle$

$p \in pred(n),\ p = \langle A_p, C_p \rangle$

$delete(P, n) \equiv$ **if** $empty(n) \vee unreachable(n)$ **then** P' **else** P

where:

$P' = \langle N', n_0', F \rangle$

$N' = N - \{n\}$

if $\neg unreachable(n)$ **then**

let $succ(n) = \{n'\}$

$$n_0' = \begin{cases} n' & \text{if } n = n_0 \\ n_0 & \text{otherwise} \end{cases}$$

$p = \langle A_p, C_p' \rangle$ where $C_p' = replace\text{-}edge(C_p, n, n')$

Figure 3.7: Definition of delete.

move-op: Pgm $\times$ Assign $\times$ $\mathcal{P}$(Node) $\times$ Node $\rightarrow$ Pgm

Let $P = \langle N, n_0, F \rangle$

$m = \langle A_m, C_m \rangle$

$\textit{move-set} \subseteq \textit{succ}(m)$

where $\langle A, C \rangle \in \textit{move-set} \Rightarrow x \in A$

$\textit{move-op}(P, x, \textit{move-set}, m) \equiv$ **if** $\textit{Cond}$ **then** P' else P where:

$\textit{Cond} \equiv \forall n_j \in \textit{move-set},\ n_j \notin F \wedge \textit{no-read-conflict}(x, n_j) \wedge$

$\textit{no-write-conflict}(x, m) \wedge \forall l, p\ \textit{dead}(l, p)$

where $p \in (\textit{succ}(m) - \textit{move-set})$ and $l \in \textit{write}(x, s)$ for some s

$P' = \langle N', n_0, F \rangle$

$N' = N \cup \{n'_j | n_j \in \textit{move-set}\}$

$m = \langle A'_m, C'_m \rangle$ where

$A'_m = A_m \cup \{x\},\ C'_m = \textit{replace-edge}(C_m, \ldots, n_j, n'_j, \ldots)$

$n'_j = \langle A'_j, C_j \rangle$ where $A'_j = A_j - \{x\}$

Figure 3.8: Definition of move-op.

the same location—recall that all reads are performed before any writes—and, therefore, it is illegal move x from n if there is another statement in n that reads the location x writes.

Lemma 3.3.2 Move-op is correct.

Proof: Let $P' = \mathit{move\text{-}op}(P, x, \{n_j\}, m)$. Clearly, if P' does not enter the region of the program modified by the transformation, then P' behaves exactly as P. Thus the traces $T(P, s)$ and $T(P', s)$ are identical up to the step that P' first executes either n or m. There are three cases:

1. Both programs execute n_j. In this case the two programs execute the same node and execution continues outside the affected region.

2. P' executes m and then n'_j; P executes m and then n_j. Because of the condition placed on the transformation, the effect of the assignments in P and P' is identical.

3. P' and P execute m and the next node is not an n'_j. In this case whatever x writes in P' is dead—see the transformation condition—and has no effect on the computation (note that x is useless in this case).

Proceeding by induction on the number of times P' enters the transformed part of the program, in a final state the stores of the two programs differ in locations that are dead. As the final statement reads the entire store (i.e., no locations are dead) in both traces, we conclude that $R(P, s) = s' \Leftrightarrow R(P', s) = s'$.[2] □

Move-test

The formal definition of move-test is in Figure 3.9. A picture is given in Figure 3.4.

[2]Recall that it is not necessary for correctness that the entire final store be the result of of a program; this approach has been adopted for simplicity.

move-test: Pgm $\times$ Test $\times$ Node $\times$ Node $\rightarrow$ Pgm

Let $P = \langle N, n_0, F\rangle$

$m = \langle A_m, C_m\rangle$

$C_m = \langle B_m, \mathit{select}_m, r_m, H_m\rangle$

$n = \langle A_n, C_n\rangle$

$C_n = \langle B_n, \mathit{select}_n, r_n, H_n\rangle$

$\{x_j\}$ = a subset of copies of x in B_n.

$\mathit{move\text{-}test}(P, \{x_j\}, n, m) \equiv$ **if** $\mathit{no\text{-}write\text{-}conflict}(x, m)$ **then** P' **else** P

where:

$P' = \langle N', n_0, F\rangle$

$N' = N \cup \left\{n_t, n_f\right\}$

$m = \langle A_m, C'_m\rangle$ where:

$C'_m = \langle B'_m, \mathit{select}'_m, r_m, H'_m\rangle$

$H'_m = H_m - \{n\} \cup \{n_t, n_f\}$

$B'_m = B_m \cup \{x\}$

$\mathit{select}'_m = \mathit{select}_m \left[\langle x, \mathrm{tt}\rangle \leftarrow n_t, \langle x, \mathrm{ff}\rangle \leftarrow n_f, \langle z, \mathrm{b}\rangle \leftarrow x\right]$

where $\mathit{select}_m(z, b) = n$

$n_t = \langle A_n, C'_n\rangle$ where:

$C'_n = \mathit{reach}(\langle B_n, \mathit{select}_n\,[\ldots, \langle z, b\rangle \leftarrow \mathit{select}_n(x_j, \mathrm{tt}), \ldots], r_n, H_n\rangle, r_n)$

where $\mathit{select}_n(z, b) = x_j$

$n_f = \langle A_n, C''_n\rangle$ where:

$C''_n = \mathit{reach}(\langle B_n, \mathit{select}_n\,[\ldots, \langle z, b\rangle \leftarrow \mathit{select}_n(x_j, \mathrm{ff}), \ldots], r_n, H_n\rangle, r_n)$

where $\mathit{select}_n(z, \mathrm{b}) = x_j$

Figure 3.9: Definition of move-test.

Lemma 3.3.3 Move-test is correct.

Proof: Let $P' = \textit{move-test}(P, \{x\}, n, m)$. As with move-op, P' behaves exactly as P up to the step that it first enters the region affected by the transformation. At that point there are four cases:

1. Both execute n. In this case the two programs execute the same node and execution continues outside the affected region.

2. P' and P execute m and the next node is not n. Test x in m is not on any paths through m's dag of tests except the paths to n_t and n_f. Thus, the evaluation of x is useless in this case.

3. P' executes m and then n_t; P executes m and then n. The nodes n_t and n differ at one point in the dag of tests: n has x at that point while n_t has the subdag rooted at x's true branch in n. Thus, if x evaluates to true in n, then the next node must be the same in each trace. By the condition of the transformation, because x evaluates to true at m in P', x evaluates to true at n in P.

4. P' executes m and then n_f; P executes m and then n. This is symmetric to the previous case.

Proceeding by induction on the number of times P' enters the transformed part of the program, we see that the traces execute the same assignments at each step, and therefore the stores are always equal. Thus the final stores must be equal and the programs are strongly equivalent. □

Unrolling

The formal definition of unrolling is given in Figure 3.10. A picture is given in Figure 3.5. In the definition, the superscript of a statement is the unrolled iteration to which the statement belongs, while the subscript distinguishes statements unroll: Pgm $\times$ $\mathcal{P}$(Node) $\times$ Int $\rightarrow$ Pgm

Let $P = \langle N, n_0, F \rangle$

$M = \{m_1, \ldots m_k\}$ be a set of nodes, satisfying:

M is a loop of P

m_1 is the unique entry point to M

$m_1 \in \mathit{succ}(m_k)$ and (m_k, m_1) is the unique backedge of M

$\mathit{unroll}(P, M, i) = P'$ where:

$P' = \langle N', n_0, F \rangle$

$N' = N \cup \bigcup_{1 \leq a \leq i, 1 \leq b \leq k} \{m_b^a\}$

Let $m_b^a = \langle A_b, C_b^a \rangle$ where:

$\forall a, b$ s.t. $1 \leq a \leq i \wedge 1 \leq b < k$,

$C_b^a = \mathit{replace\text{-}edge}(C_b, m_1, m_1^a, m_2, m_2^a, \ldots)$

$\forall a$ s.t. $1 \leq a < i$ $C_k^a = \mathit{replace\text{-}edge}(C_b, m_1, m_1^{a+1}, m_2, m_2^a, m_3, m_3^a, \ldots)$

$C_k^i = \mathit{replace\text{-}edge}(C_b, m_2, m_2^i, m_3, m_3^i, \ldots)$

Figure 3.10: Definition of unrolling.

within a loop body. The use of *replace-edge* ensures that a node's successors are from the same unrolled iteration or, if a backedge is being replaced, that the node points to the first node of the next unrolled iteration. The correctness of unrolling follows from the following lemma.

Lemma 3.3.4 Let $P' = unroll(P, \{m_h\}, i)$. Then $\langle n_0, s_0\rangle \xrightarrow{*} \langle n_j, s_j\rangle$ in P implies that $\langle n'_0, s_0\rangle \xrightarrow{*} \langle n'_j, s_j\rangle$ in P' where:

$$n_j = n'_j \vee (n_j = m_h \wedge n'_j = m^g_h)$$

Proof: The proof is by induction on the length of $T(P, s)$ and $T(P', s)$.

Basis: Trivial.

Step: Assume the lemma holds for $\langle n_0, s_0\rangle \xrightarrow{*} \langle n_j, s_j\rangle$ and $\langle n'_0, s_0\rangle \xrightarrow{*} \langle n'_j, s_j\rangle$. There are two cases:

1. $n_j = n'_j$. Then n_{j+1} and n'_{j+1} satisfy the lemma because the nodes are identical and evaluated in the same store.

2. $n_j = m_h \wedge n'_j = m^g_h$. The nodes contain identical statements; only the names in the dags are different. Either a name was not replaced, in which case $n_{j+1} = n'_{j+1}$, or it was replaced, in which case $n_{j+1} = m_{h'} \wedge n'_{j+1} = m^{g'}_{h'}$. This satisfies the requirements of the induction.

□

Chapter 4

Properties of the Core Transformations

In this chapter we explore properties of our transformational system. We show that two desirable properties of transformational systems, *confluence* and *completeness*, do not hold for our system. Analysis of the difficulties with confluence and completeness shows that the most important problem is the presence of false dependencies in programs. False dependencies are an inherent property of imperative languages; therefore, we argue that any interesting parallelization system for imperative languages cannot be be confluent or complete. For a restricted class of programs with no false dependencies, we show that our transformational system is complete.

We would like to show that our transformational system is expressive—that the transformations can significantly improve a program. The restricted completeness property, combined with the fact that there are very good techniques for eliminating false dependencies in practice, is evidence that the core transformations are expressive. In the remainder of this dissertation we focus on particular algorithms (and families of algorithms) built from the core transformations, and show that those algorithms achieve the effect of many parallelizing transformations in the literature. In this way, we show that the combination of a small number of simple transformations is at least as expressive as a large body of existing transformations.

4.1 Confluence

We write $P \mapsto P'$ if P' can be obtained from P by a (possibly empty) sequence of core transformations.

Definition 4.1.1 (Confluence) A transformational system is *confluent* if

$$A \mapsto B \wedge A \mapsto C \Rightarrow \exists D \text{ s.t. } B \mapsto D \wedge C \mapsto D$$

(Confluence is also called the diamond property [Bar84].)

Confluence shows that the final result of a sequence of transformations is independent of the choice of transformation at each step in the sequence. This is a desirable property—if a system of transformations is confluent, then there is no concern that a sequence of transformations can lead to a local optimum from which it is impossible to proceed to a global optimum.

Unfortunately, the core transformations are not a confluent system. The three programs in Figure 4.1 provide a counter-example. Program A transforms to both B and C as shown. In program B, the only possible transformation is a move-op applied to $a := 1$. However, this fails due to a false dependency caused by the assignment to a in the first node. By a symmetric argument, no transformations are possible on program C. Thus, there is no program D such that $B \mapsto D$ and $C \mapsto D$. The system is not confluent because moving a statement past a branch may introduce new false dependencies which prevent other statements from moving.

No interesting parallelization system is confluent precisely because imperative programs have false dependencies. The problem can be handled in a variety

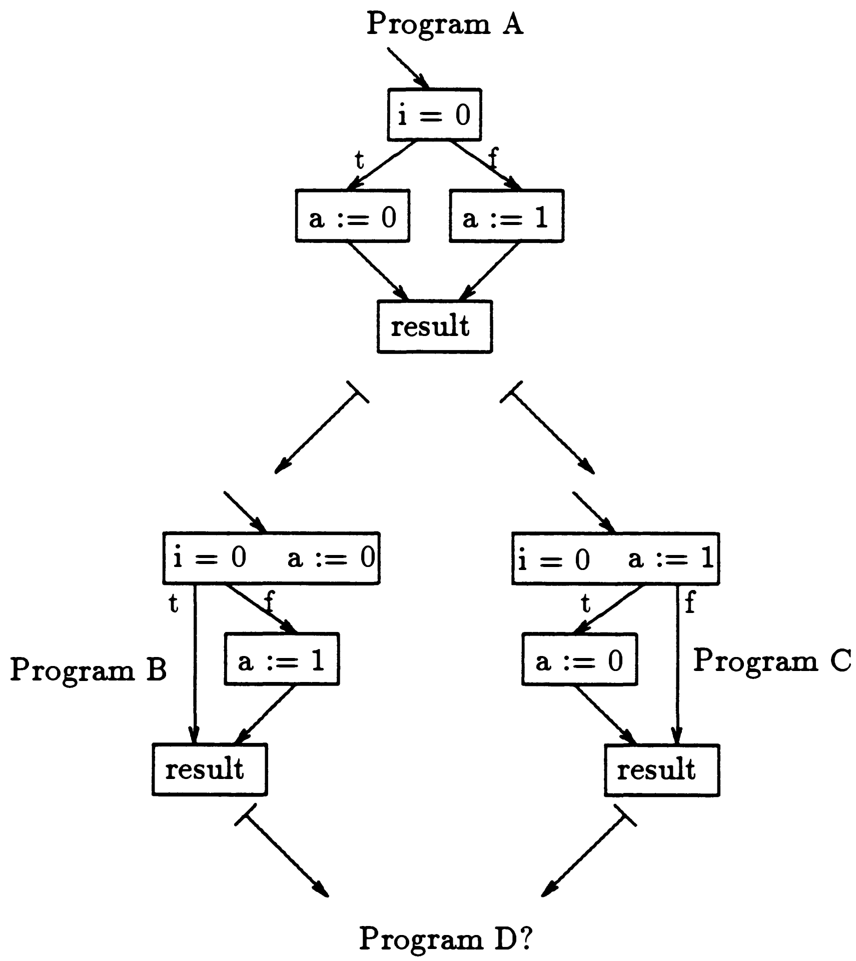


Figure 4.1: The core transformations are not confluent.

of ways. All systems use renaming (substitution of new variable names) to remove as many false dependencies as possible. Trace scheduling uses heuristics to estimate which paths in a program are most frequently executed; priority for code motion is given to statements on frequently executed paths [Fis81,Ell85]. Another approach, called *If-conversion*, is to eliminate all flow of control from a program by adding appropriate guards to statements [AKPW83]. In practice, most false dependencies can be eliminated [Wol78,KKP+81,CF87].

4.2 Completeness

Another desirable property of a transformational system is completeness. A parallelizing system is complete if for any P' that is a parallelization of P, P' is achievable from P in the system. In our case, the model of parallelization is $\leq_p$, so a reasonable definition of completeness could be:

$$P \leq_p P' \Rightarrow P \mapsto P'$$

This definition is, however, too strong. For example, consider Figure 2.7; $P \equiv_p P'$ but there is no sequence of transformations such that $P \mapsto P'$. Another example is given in Figure 4.2; $P \equiv_p P'$, but P does not transform to P' because no transformations reorder the tests within a node.

With this definition of completeness, the core transformations are incomplete simply because there is more than one way to write the same program. The following definition captures the idea that a system is complete if for any parallelization P' of P, something at least as good as P' is achievable from P in the system.

Definition 4.2.1 (Completeness) A system of parallelizing transformations is *complete* for a program P if for every P' such that $P \leq_p P'$ there exists a P'' such that $P \mapsto P''$ and $P' \leq_p P''$. An equivalent statement is that for every P' such that $P \leq_p P'$, there is a P'' such that the diagram in Figure 4.3 commutes.

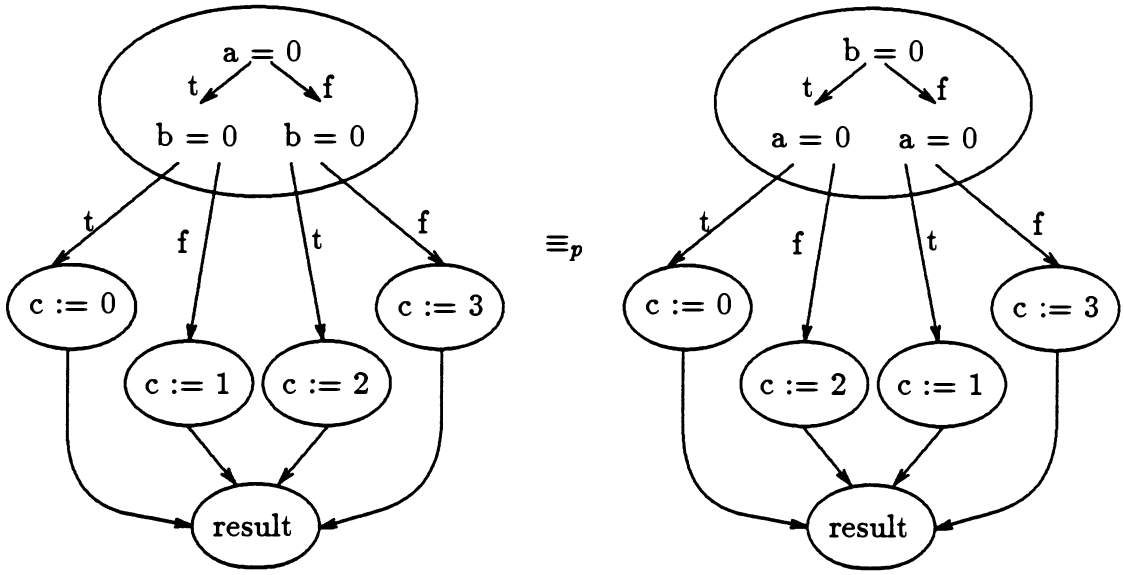


Figure 4.2: Another example of incompleteness.

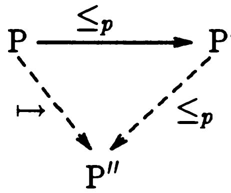


Figure 4.3: Definition of completeness.

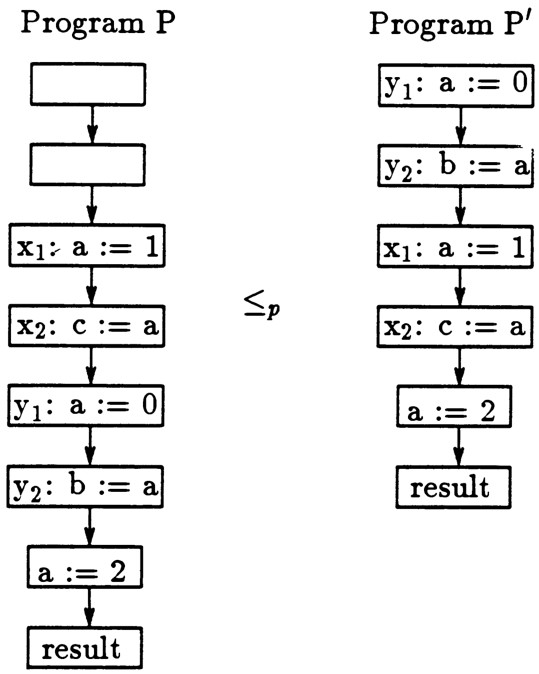


Figure 4.4: The core transformations are incomplete.

Completeness is desirable because it shows that, with respect to the model, no other transformational system is more powerful. The core transformations are not complete. An example is given in Figure 4.4. In this figure, $P \leq_p P'$ because all of the true dependencies of P are preserved by P'. The only difference between P and P' is the position of statements y_1 and y_2. Because y_2 depends on y_1, the first transformation in any sequence converting P into P' must move y_1. However, y_1 cannot move past x_2 because of the false dependency with x_2.

This example shows that completeness requires non-local transformations. Statements x_1 and x_2 can be interchanged with statements y_1 and y_2, but they cannot overlap. Despite this result, we have not included non-local transformations in our system for two reasons. First, recognizing and performing a non-local transformation is much more expensive than recognizing and performing a core transformation. Second, we have not found a real example where this problem cannot be eliminated by renaming techniques. In Figure 4.4, choosing a new name for a in statements y_1 and y_2 eliminates the problem. To our knowledge, no existing parallelization system incorporates non-local transformations of the type in Figure 4.4; it is always assumed that false dependencies must be observed.

Renaming cannot always eliminate false dependencies if the programming language has aggregate data structures (e.g., arrays). An example is given in Figure 4.5. For clarity, only data dependency edges are shown. The flow of control is from top to bottom. Solid lines represent true dependencies; broken lines represent false dependencies. The possible values of the subscript variables are given at the bottom of the figure. It is assumed that this is a fragment of a larger program in which the subscript variables can have only these values and the value of a variable is independent of the values of other variables. In this example, the false dependency cannot be eliminated by renaming the array A, since all references to A must be given the same name. Because of the false dependency, statements w_3 and r_2 can be interchanged with statements w_2 and r_1.

In [CF87], Cytron and Ferrante present an algorithm for removing all false dependencies from a program. Variables are renamed and new statements—which essentially perform runtime dependency testing—are added to the program. Using this algorithm, all false dependencies can be removed from the example in Figure 4.5. However, this approach may not result in a better program. Let P' be the result of applying Cytron and Ferrante's algorithm to P. The program P' is generally a slower program than P, because extra statements have been introduced to eliminate false dependencies. Thus, it is not necessarily the case that parallelizing P' produces a faster program than parallelizing P, even though the speedup achieved from parallelizing P' will probably be greater than the speedup achieved from parallelizing P.

w_1: $A[i_1] := a;$

w_2: $A[i_2] := b;$

r_1: $c := A[i_3];$

w_3: $A[i_4] := d;$

r_2: $e := A[i_5];$

where $i_1 = 0$ or 1
$i_2 = 2$
$i_3 = 0$ or 2
$i_4 = 2$
$i_5 = 1$ or 2

Figure 4.5: Simple renaming cannot eliminate all false dependencies.

4.3 A Completeness Result

In this section, we show that the core transformations are complete for *tree programs* without false dependencies. This shows that the most important source of incompleteness of the core transformations is the presence of false dependencies. Moreover, it shows that the core transformations are an expressive system. If a program graph is a tree and all program nodes have test dags which are trees, then the program is a tree program. We assume that the statements of a tree program are useful in every computation.

Definition 4.3.1 Let P be a program. Consider any execution sequence $\langle n_0, s\rangle \xrightarrow{*} \langle n_i, s_i\rangle \xrightarrow{*} \langle n_j, s_j\rangle$. Program P is *free of false dependencies* if for every $x \in op(n_i)$ and $y \in op(n_j)$, $read(x, s_i) \cap write(y, s_j) = \emptyset$.

The relationship $P \leq_p P'$ holds if there is a dependency preserving mapping from every execution trace $T(P, s)$ of P to every corresponding execution trace $T(P', s)$ of P'. Let $\mathcal{F}$ be a set of such mappings establishing $P \leq_p P'$. Let x' be a statement in P'. We define the *inverse image* of x' under $\mathcal{F}$ to be the set

$$X = \left\{x | f(x) = x' \text{ for some } f \in \mathcal{F}\right\}$$

Lemma 4.3.2 For any useful statement x' in P', the inverse image of x' under $\mathcal{F}$ is nonempty.

Proof: By definition of **sim**. □

Lemma 4.3.3 Let $P \leq_p P'$, where P is a tree program free of false dependencies. Then $P \mapsto P^*$, where P^* is a tree program and there is a set of mappings $\mathcal{F}^*$ demonstrating $P^* \leq_p P'$ and satisfying

$$x \in exec(n_0^*, s) \Rightarrow f^*(x) = x' \in exec(n_0', s)$$

where n_0^* is the start node of $T(P^*, s)$ and $f^* \in \mathcal{F}^*$ is the corresponding dependency preserving mapping from $T(P^*, s)$ to $T(P', s)$.

Proof: Let $x' \in op(n'_0)$, and let X be the inverse image of x' under $\mathcal{F}$. By Lemma 4.3.2, X is not empty. Consider $x_i \in X$. By assumption, there is no false dependency on x_i. Furthermore, there cannot be a true dependency on x_i because $P \leq_p P'$ and $x' \in op(n'_0)$. Thus, x_i can be blocked from moving into a node m only because it writes a value that is live on an exit from m. For brevity, we say that x_i is blocked by a *live conflict* at m in this case. Subject to live conflicts at branch points on the paths from $node(x_i)$ to n_0, every x_i can move to n_0. By combining the statements in X at common ancestors as they move through the program, a single copy x can move to n_0.

We construct a sequence of transformations without live conflicts that moves statements of P to n_0 that correspond to statements in n'_0 of P'. In general, statements in n'_0 may be false dependent on one another; the definition of completeness only requires P to be free of false dependencies. Choose an $x' \in op(n'_0)$ such that x' is not false dependent on any statement of $op(n'_0)$. Repeatedly move an $x_i \in X$ that is deepest in the program graph; this allows multiple copies to be combined at common ancestors. As stated previously, no x_i can be stopped by a true or anti-dependency.

The only possible problem is a live conflict; this situation is shown in Figure 4.6a. In this example, x_i cannot move because it writes a value that statement y might read. But there is a mapping showing $P \leq_p P'$ where x_i is mapped to x' in n'_0. By assumption, x_i is not false dependent on any statement in n'_0, so statement y in P' must be in a node other than n_0. It cannot be that y is dependent on x in P', or else P is not **sim** to P'. Therefore, P' must have the structure shown in Figure 4.6b, where some other statement z prevents y from reading the value x writes. But where is z in program P? It cannot be on the path from n to m, or else there is no live conflict. It cannot be on the path from n_0 to n or else x_i is false dependent on z and P is not free of false dependencies. Therefore, if z is useful, then z is not in P, implying P is not **sim** to P'—a contradiction.

We conclude that no live conflict can occur.

Inductively, assume that all sets X in P corresponding to an x' that are false dependent on a chain of at most $j-1$ statements in n'_0 of P' have been moved to n_0 in P. The previous argument carries through without modification for statements in n'_0 that are false dependent on a chain of j other statements in n'_0.

We have shown that $P \mapsto P^*$, where P^* has the same statements in n^*_0 as P' has in n'_0. Note that $\mathcal{F}$ can be modified to map statements in n^*_0 to statements in n'_0, giving the required $\mathcal{F}^*$. □

Lemma 4.3.4 Let P be a tree program free of false dependencies. For any program P' such that $P \leq_p P'$, there is a P'' such that $P \mapsto P''$, $P'' \leq_p P'$, and for any pair of execution traces $T(P', s)$ and $T(P'', s)$ there is a dependency preserving mapping f satisfying:

$$x \in op(n_i) \Rightarrow f(x) \in op(n'_i)$$

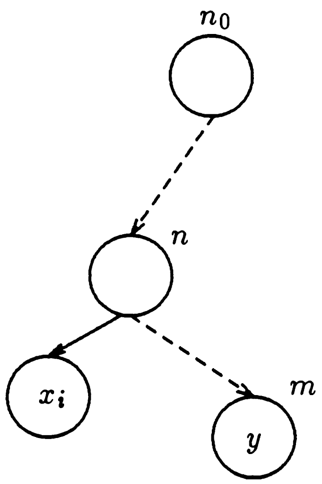
where $\langle n_i, s_i \rangle$ is the ith step of $T(P'', s)$ and $\langle n'_i, s'_i \rangle$ is the ith step of $T(P', s)$.

Proof: The proof is by induction on the height of the tree program P.

Basis: The height of the program graph is one. There is only one such program (a single node containing *result*), so the lemma holds trivially.

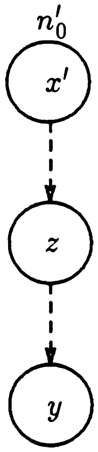
Step: Assume the lemma holds for tree programs of height $i-1$ or less. By Lemma 4.3.3, $P \mapsto P^*$ where there are functions $\mathcal{F}^*$ showing $P^* \leq_p P'$ that map statements in n^*_0 to statements in n'_0.

Let m' be the node name terminating any path through the test dag of n'_0. Because $P^* \leq_p P'$, this corresponds to a set of node names $\{m_i\}$ terminating a set of paths through the test dag of n^*_0. Each m_i can be viewed as the start node of a tree program free of false dependencies of height $i-1$. Let P'_m be the subgraph of P' rooted at m', and let P_{m_i} be the subtree rooted at m_i in P^*. Because statements in n^*_0 are mapped to statements in n'_0 and $P^* \leq_p P'$, it



where x writes a location y may read

(a) A live conflict in P ...



where $z \prec y$

(b) ... implies there is a statement in P'

Figure 4.6: Construction for the completeness proof.

follows that $P_{m_i} \leq_p P'_m$. The inductive hypothesis applies to each subprogram P_{m_i} corresponding to a program P'_m where m' terminates a path in the test dag of n'_0. The inductive cases are independent: the fact that the test dag of n^*_0 is a tree and $P^* \leq_p P'$ imply that no P_{m_i} is included in more than one set corresponding to an m'.

The only remaining point is that when all statements have been moved, there may be empty nodes at the end of the tree program. When these are deleted, occurrences of *result* in the final program P'' must have the same position in every trace of P'' as occurrences of *result* in corresponding traces of P'. □

Theorem 4.3.5 The core transformations are complete for tree programs without false dependencies.

Proof: Let P be a tree program without false dependencies and let P' be a program such that $P \leq_p P'$. By Lemma 4.3.4 $P \mapsto P''$ where there is a set of functions $\mathcal{F}$ showing $P'' \leq_p P'$ and satisfying

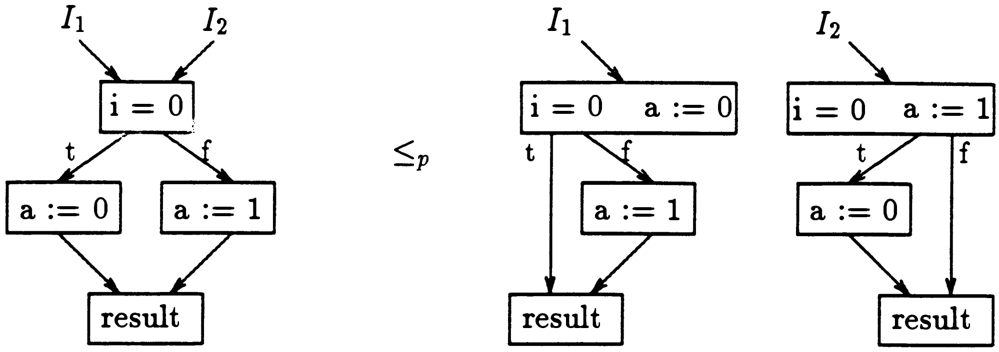
$$\forall s \, \exists f \in \mathcal{F} \text{ s.t. } pos(x) = i \text{ in } T(P, s) \Rightarrow pos(f(x)) = i \text{ in } T(P', s)$$

By definition of **sim**, these mappings are invertible, so $P' \leq_p P''$. □

4.4 Discussion

The core transformations are neither confluent nor complete. This is primarily because imperative programs have false dependencies—dependencies that represent the reuse of storage rather than real ordering constraints.

For the restricted class of tree programs without false dependencies the core transformations are complete. This completeness result is probably the best possible within the framework of $\leq_p$. Even for a simple extension, acyclic programs, the core transformations are not complete. An example is given in Figure 4.7.



Program P Program P'

Figure 4.7: Incompleteness for acyclic programs.

In program P, the fragment of the program has two entry points. Assume that dependencies are such that no statement can move out of the depicted region. In program P', there are two copies of the fragment, and the two copies are parallelized in different ways. Because the core transformations are not confluent, there is no P'' such that $P \mapsto P''$ and $P' \leq P''$. This is an artificial example; it is hard to imagine a case where P' would be the desired parallelization of P. However, there does not seem to be a natural restriction that eliminates this case.

The completeness result for tree programs is interesting for two reasons. First, this result shows that false dependencies are the most important cause of incompleteness—the example in Figure 4.4 shows the core transformations are not complete for any non-trivial class of programs in the presence of false dependencies. False dependencies arise from the von Neumann model of computation with its persistent state; for this reason, many people have argued that imperative languages are undesirable for compilation, among other things. (See, for example, [Bac78,AI83].) To our knowledge, this is the first formal result showing that when false dependencies are not present, the expressive power of a program transformation system increases.

Second, this result provides evidence that the core transformations are expressive, especially given the fact that good techniques exist for eliminating false dependencies in practice. Completeness is a strong property: at least in this domain [illegible] ansformational system is more powerful than the core transformations.

Chapter 5

Higher-Level Transformations

This chapter develops compaction algorithms based on the core transformations. In the first section, we present an algorithm, *migrate*, for moving a single statement as far "up" as possible in the program graph. We show that migrate is the best strategy for moving a single statement. In the second section, we show that the core transformations can implement *trace scheduling* [Fis79,Fis81]. In the third section, migrate is used to define a generalization of trace scheduling, *compact-global*.

5.1 Migrate

Migrate moves a statement as far in the program graph as dependencies allow. Recall that a statement moved by the core transformations may be duplicated to preserve program semantics; thus, a good algorithm for moving a statement should move copies of the statement that are created during compaction. The problem is to move copies in an order that allows as many copies as possible to be combined[1] at common ancestors in the program graph. Opportunities for combining copies of a statement arise when a statement is copied on different paths–perhaps several times–and then at least some of the copies can move to

[1] "Unified" in [Nic85b,AN88a].

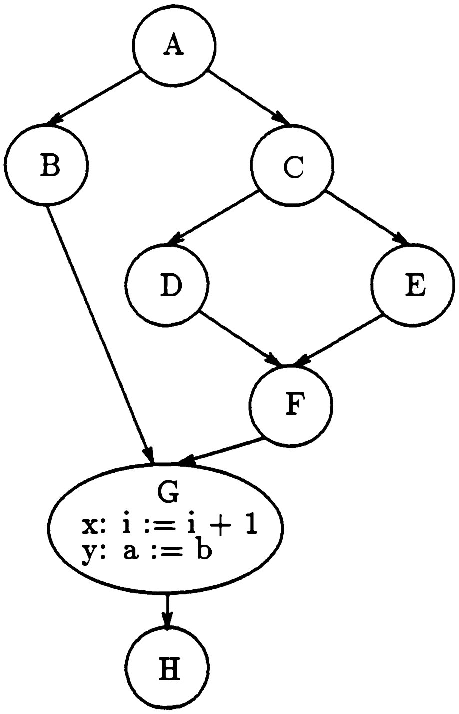


Figure 5.1: A sample program.

the point where the paths rejoin.

For example, assume in Figure 5.1 that no statements in the depicted region except x and y read or write variables a, b, or i, so statements x and y can move on all paths to node A. Figure 5.2 shows the program after the transformation *move-op*(x, G, B). At this point, statement x in node B cannot move into node A because of the copy of x in node G. If x in G can move to node C, then the two copies can move together into A.

Now consider the graph after transformations *move-op*(y, G', B) and *delete*(G'). The resulting program is shown in Figure 5.3. Unlike statement x, statement y in node B can move immediately into node A, because a is dead on the other exit from A. However, this would leave an unnecessary copy of y in G. (Note that the copy of y in G would be redundant because A dominates G.) As before, if y in G first moves to node C, then the two copies can move together into A.

We restrict migrate to acyclic program graphs. Loops are treated as acyclic

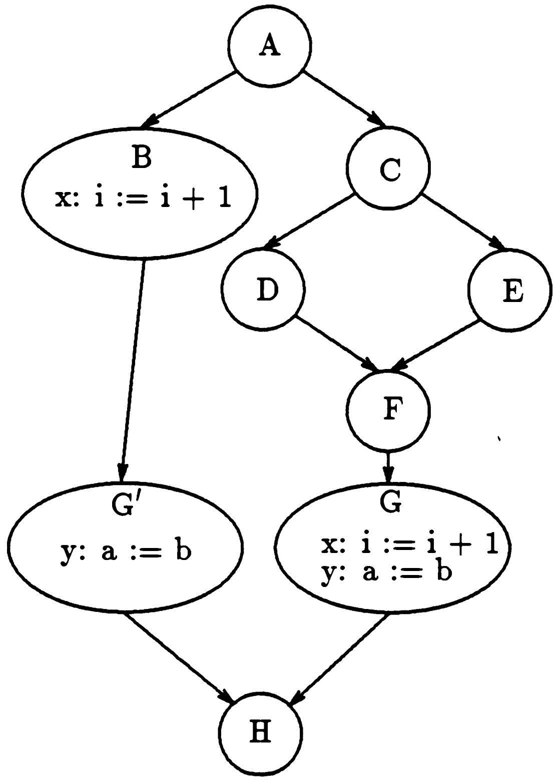


Figure 5.2: The program after x is moved to B.

graphs by disallowing transformations across the backedges of a loop. (A loop is a cycle in the control-flow graph.) The only advantage of moving statements across backedges is that consecutive loop iterations can be overlapped; this can also be accomplished by unrolling the loop and compacting within the unrolled loop body (see Chapter 6).

The restriction to acyclic graphs makes it possible to define an optimal migrate algorithm; no algorithm for moving a single statement in a loop is optimal. To see this, consider the special case of moving a *loop-invariant* statement.[2] When a statement is moved from an entry node of a loop to a node outside of the loop, the statement is copied on the edge inside the loop body. A loop-invariant statement can move an arbitrary number of times inside the loop, so no finite sequence of transformations can be called optimal. An example of this situation is given in Figure 5.4.[3] We show that an algorithm for moving a single statement and its copies in an acyclic program is optimal if it satisfies the following invariant.

Constraint 5.1.1 (Migrate Invariant) Whenever $\textit{move-op}(x, \{n_i\}, m)$ is performed, there is no m' reachable from m such that $\textit{move-op}(x, \{n'_i\}, m')$ can be performed.

Figure 5.5 gives the migrate algorithm for assignments. Define the *depth* of node n to be the length of the longest path from the start node to n. The migrate algorithm repeatedly considers all possible transformations $\textit{move-op}(x, \{n\}, m)$ and executes one that maximizes the depth of m. We show that this satisfies Constraint 5.1.1.

Lemma 5.1.2 The migrate algorithm satisfies Constraint 5.1.1.

Proof: Let $\textit{move-op}(x, \{n\}, m)$ be a transformation selected by migrate and assume it does not satisfy the invariant. Then there is another possible transforma-

[2]A loop-invariant statement depends on no statement in the loop, including itself [ASU86].

[3]A good compiler removes loop-invariant statements from a loop [ASU86]; this example is used to motivate the restriction to acyclic graphs.

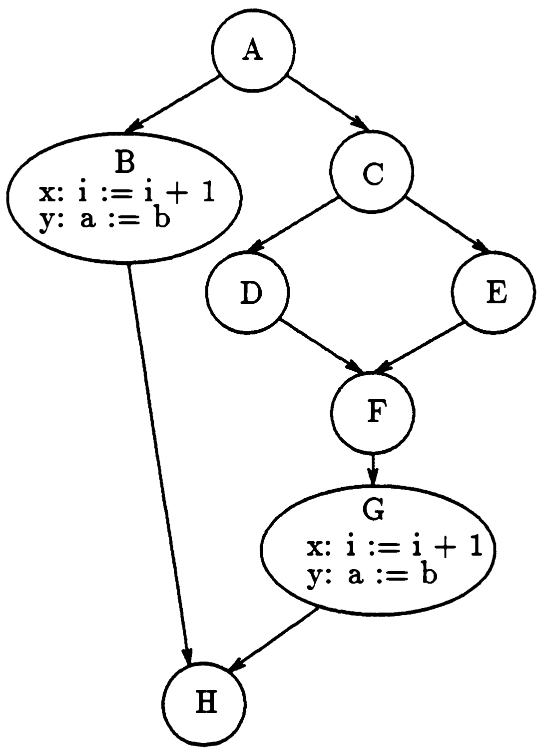


Figure 5.3: The program after y is moved to B.

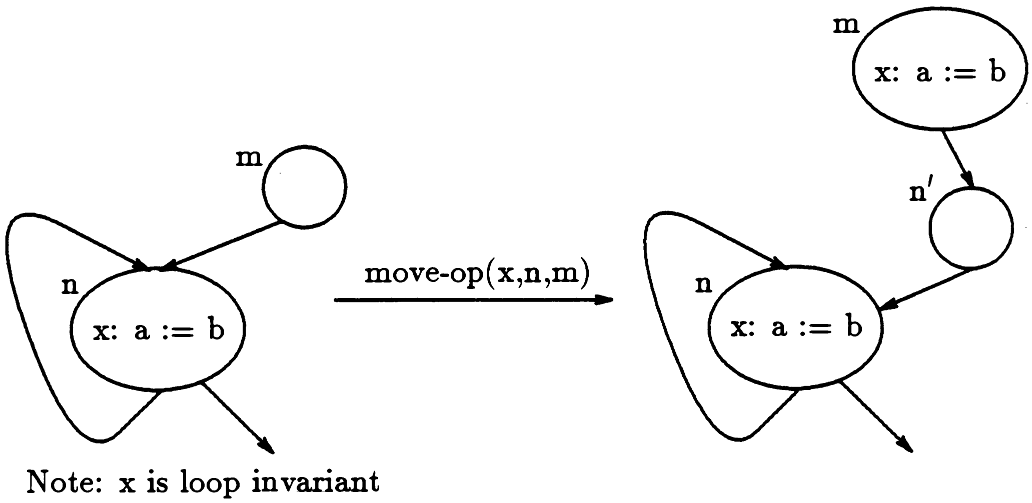


Figure 5.4: An anomalous situation for migrate.

```
(* x is an assignment *)
X := {x};
repeat
    (* We assume that X always contains all copies of statement x *)
    Select x ∈ X such that x can move to a predecessor m of node(x)
        and the depth of m in the graph is maximized;
    move-op(x, move-set, m);
        where move-set is the largest subset of succ(m) where x can
            move from every node to m.
until no statement in X can move;
```

Figure 5.5: The migrate algorithm for assignments.

```
(* x is a test *)
X := {x};
repeat
    (* We assume that X always contains all copies of statement x *)
    Select x ∈ X such that x can move to a predecessor m of node(x)
        and the depth of m in the graph is maximized;
    move-test(move-set, node(x), m);
        where move-set contains all copies of x in node(x).
until no statement in X can move;
```

Figure 5.6: The migrate algorithm for tests.

tion $move\text{-}op(x, \{n'\}, m')$ where m' is a successor of m, so $depth(m') > depth(m)$. But the algorithm chooses a transformation that maximizes the depth of m, a contradiction. □

The intuition behind the algorithm is that every transformation moves the copies that are deepest in the program graph, thus ensuring that a statement moves only after any statement it could be combined with has moved. Each node in the graph is labelled with its depth using a modified depth-first search [AHU74]. When a node is copied, it inherits the depth label of the original node. While this may not be the actual depth of the copied node in the program graph, it is a consistent labelling: all ancestors of the copied node have lesser labels and all successors have greater labels.

We show that migrate is an optimal algorithm for moving a single assignment and its copies. Let $C(x,t)$ denote the set of all copies of the original statement x after t transformations. The following technical lemma is needed for the proof of optimality.

Lemma 5.1.3 Let $T_1, T_2, \ldots, T_n$ be the sequence of transformations performed by migrate. If $x' \in C(x,n)$ is blocked only by a live conflict at m, then the statement y causing the live conflict does not depend on a copy of x.[4]

Proof: The proof is by induction on the number of transformations.

Basis: No transformations are performed, so the lemma is trivially satisfied.

Step: Assume the lemma holds after i transformations. Let transformation T_{i+1} move copies of x to m. By the invariant, no other copy of x in a node reachable from m can move. Thus, after T_{i+1} the lemma holds for statements in nodes reachable from m. Furthermore, T_{i+1} has no effect on the transformations possible for statements in nodes not reachable from m, so the lemma holds for these statements as well.

[4]Recall that x is blocked by a live conflict at m if there is a statement y reachable from m on another path such that x writes a value y reads.

Assume that the lemma does not hold for x' in m after transformation T_{i+1}. Then there is an x'', a copy of x, in a predecessor p of m that writes a value read by y in a successor of p. But $depth(m) > depth(p)$, so by the invariant no copy of x can move past p before transformation T_{i+2}, and the lemma holds. □

Theorem 5.1.4 Migrate is an optimal algorithm for moving a single assignment and its copies:

$$\forall P, x\ \mathcal{M}'(P, x) \leq_p \mathcal{M}(P, x)$$

where $\mathcal{M}$ is migrate, $\mathcal{M}'$ is any other algorithm that moves only x and its copies, and P is an acyclic program.

Proof: Consider two execution traces $T(\mathcal{M}(P,x),s)$ and $T(\mathcal{M}'(P,x),s)$. We construct a mapping f showing that $pos(y)$ in $T(\mathcal{M}'(P,x),s)$ is greater than or equal to $pos(f(y))$ in $T(\mathcal{M}(P,x),s)$.

All statements except copies of x have the same position, so the obvious one-to-one mapping suffices for these statements. No core transformation (or sequence of transformations) adds a useful statement to or removes a useful statement from an execution trace (see Definition 2.3.12), so the execution traces have the same number of useful copies of x as the original program. Thus, they have one useful copy of x each, the last one appearing in the execution trace. Let x' be the useful copy of x in $T(\mathcal{M}'(P,x),s)$ and let x'' be the useful copy of x in $T(\mathcal{M}(P,x),s)$. We show that $pos(x') \geq pos(x'')$.

Assume that $pos(x') < pos(x'')$. Let m be the predecessor of $node(x'')$ on a path to the position of x'. From the conditions of move-op, there are five ways that x'' can be prevented from moving to m:

1. $node(x'')$ is the start node.

2. x'' (false) depends on $y \in op(node(x''))$.

3. x'' (true) depends on $y \in op(m)$.

4. x'' writes a value read by a statement y on another path from m, where y is not a copy of x.

5. x'' writes a value read by a statement x''' on another path from m, where x''' is a copy of x.

By construction, when migrate terminates no further transformations are possible. Therefore, one of the five cases must apply to x''. If x'' is in case 1, 2, or 3, then x' could not have moved past the same point in the program. If x'' is in case 4, then, by Lemma 5.1.3, x'' is blocked by a live conflict involving only statements that are not copies of x, so again x' could not have moved past the same point in the program. So x'' must be in case 5. Consider what prevents x''' from moving. As with x'', x''' cannot be in cases 1–4, or else x''' could not have moved past that point in the program in any sequence of transformations, which implies that x'' could not have moved further in any sequence of transformations because of its live conflict with x'''. So x''' is also in case 5. Furthermore, there must be a statement causing the live conflict with x''' other than x'', or else move-op could combine the two statements. Iterating this procedure, we can construct an arbitrarily large set of distinct copies of x, each blocked by a live conflict with another member of the set. But the program has a finite number of statements, so no such set exists. Therefore, $pos(x'') \geq pos(x')$. $\square$

A migrate algorithm for moving tests is given in Figure 5.6. A slightly different algorithm is required because move-test combines multiple tests from within in a single node, while move-op combines assignments from multiple nodes. The proof of optimality for migrate with tests is very similar to the proof for migrate with assignments.

5.2 Trace Scheduling

In this section, we implement *trace scheduling* with the core transformations [Fis81]. Using the core transformations as primitives leads to a simpler and more comprehensible algorithm than those presented in [Fis81,Nic84,Ell85]. In Section 5.3, we use this derivation as the basis for a generalization of trace scheduling.

We have defined the term "execution trace" to be an execution history of a program. Trace scheduling defines a "trace" to be a path through a program graph. To avoid confusion, we use the terms "execution trace" for an execution history and "path" for a trace scheduling trace wherever necessary.

Trace scheduling is based on the assumption that, in practice, the execution of one (or a few) paths dominates the computation time of programs, and especially of scientific programs. For the purposes of this chapter, we accept this assumption.[5]

Trace scheduling repeatedly selects an acyclic path through the program graph and compacts it. Trace scheduling tries to compact the "most important" paths first. This is done by heuristically assigning probabilities $true(x)$ and $false(x)$ to each test x; these represent the likelihood that the true and false branches of x are taken respectively. A special procedure, the *trace picker*, iteratively selects an uncompacted path with highest probability of execution. An outline of the trace picker is given in Figure 5.7. In the figure, $prob(n)$ is a priority function on nodes based on the probabilities assigned to tests.

Using the trace picker, trace scheduling repeatedly selects a path and compacts that path as much as possible. Figure 5.2 gives an algorithm implementing trace scheduling with the core transformations. The procedure *migrate-within-trace* moves a statement as far as possible on the path selected by the trace picker.

[5] The validity of this assumption is not universally accepted—there is some empirical evidence to the contrary [Shu77].

It is somewhat surprising that there is such a simple expression of trace scheduling. The original trace scheduling algorithm, as developed in [Fis81,Nic84, Ell85] is very complicated, with several subtle cases. In Ellis' thesis [Ell85], the description of trace scheduling, without a proof of correctness, requires twenty pages. This complexity arises from the trace scheduling view that the primitive transformation is compacting a path. With this view, determining the statements that must be copied on other paths to preserve semantics requires reasoning about all the possible orderings of statements before and after compaction. (In our version of trace scheduling, the copying is handled automatically by the core transformations.) This leads to a difficult case analysis; so difficult, in fact, that a bug in the original trace scheduling algorithm was not noticed for several years [Fis79,Nic84]. Because the core transformations preserve semantics, the algorithm in Figure 5.9 is automatically correct.

We will not attempt to prove that this derivation of trace scheduling is equivalent to the original formulation. The most important point is that the copied statements generated by trace scheduling to preserve semantics at the entrances and exits to compacted paths are exactly the statements duplicated by the transformations performed by the algorithm in Figure 5.2.

5.3 Compact-Global

Trace scheduling is overly restrictive, because trace scheduling moves statements on only a single path at a time. Theorem 5.1.4 shows that the best strategy is to move a statement on all paths. Thus, even if the most important path can be predicted, trace scheduling does not necessarily perform the best possible compaction of that path. This weakness of trace scheduling is illustrated in Figure 5.1. If statement x is moved only on one path, it cannot move to node A because it will be blocked by copies of itself. Migrate succeeds in moving x to A because it selects an appropriate order in which to move the copies of x on all

```
procedure pick-trace(P)
(* N is the set of all nodes;
    C is the set of all uncompacted nodes *)
    begin
        rest := N − C;
        select n ∈ rest where prob(n) = max_{n'∈rest} prob(n');
        T := (n);
        S := succ(n) ∩ rest;
        while ∃n' ∈ S do
            begin
                select n' ∈ S where prob(n') = max_{n''∈S} prob(n'');
                append(T, n');
                n := n';
                S := succ(n') ∩ rest
            end;
        C := C ∪ T;
        return(T)
    end;
```

Figure 5.7: The trace picker.

```
procedure move-one(x, n, m)
    if x is an assignment
        then move-op(x, n, m)
        else move-test(x, n, m);

procedure migrate-within-trace(x, T)
    begin
        let T = (n_1, ..., n_|T|);
        let node(x) be n_i;
        while (no move has failed) ∧ (i ≠ 1) do
            begin
                move-one(x, n_i, n_{i-1});
                i := i - 1
            end
    end;

procedure trace-scheduling(P)
    begin
        let P = ⟨N, n_0, F⟩;
        C := ∅;
        while C ≠ N do
            begin
                T := pick-trace;
                let T = (n_1, ..., n_|T|);
                for i := 1 to |T| do
                    for each x ∈ op(n_i) do
                        migrate-within-trace(x, T)
            end
    end;
```

Figure 5.8: The trace scheduling algorithm.

paths.

Trace scheduling does not allow statements to move across trace boundaries; once a path is compacted no statement can be moved onto that path from any other trace. Because the trace picker selects disjoint paths, after a few paths are compacted the remaining uncompacted paths are generally short. This reduces the potential parallelism that can be extracted.

Finally, trace scheduling produces much unnecessary copying. In particular, the fact that statements cannot cross trace boundaries implies that there is no way to combine copies of a statement from different paths. With migrate, there is some chance that multiple copies of a statement can rise to a common ancestor in the program graph and be combined, thus minimizing code explosion.

Compact-global is an algorithm without these limitations. The algorithm is given in Figure 5.9. Instead of moving a statement on a single path, compact-global uses migrate move a statement on all paths. In contrast to trace scheduling's procedure for moving individual statements, migrate is not a heuristic (Theorem 5.1.4). Thus, the only level at which heuristics play a role in compact-global is in selecting the order in which migrate is applied to statements. Heuristic choices at this level are unavoidable because the core transformations are not confluent (Section 4.1).[6]

Trace scheduling is based on the assumption that the significant speedup is achieved by identifying and compacting the most important path through a program. The following theorem shows that, by this measure, compact-global is a generalization of trace scheduling. We assume that both algorithms choose statements to move from within a node in the same order.

Theorem 5.3.1 Let TS_1 and CG_1 be trace scheduling and compact-global ap-

[6]Other heuristics are possible and perhaps desirable; we have selected one that represents a significant improvement of trace scheduling.

```
(* Let X be all copies of original program statement x *)
procedure compact-global(P)
    begin
        let P = ⟨N, n0, F⟩;
        C := ∅;
        while C ≠ N do
            begin
                T := pick-trace;
                let T = (n1, ..., n|T|);
                for i := 1 to |T| do
                    for each x ∈ op(n) do
                        migrate(X);
                C := C ∪ T
            end
    end;
```

Figure 5.9: The compact-global algorithm.

plied to only the first (most important) path.

$$\forall P \; TS_1(P) \leq_p CG_1(P)$$

Proof: By induction on the number of statements moved.

Basis: One statement is moved. The result is immediate by Theorem 5.1.4.

Step: Assume after $i-1$ statements are moved that $TS_1(P,x) \leq_p CG_1(P,x)$. Let x be the ith statement. If x is blocked on a path by a statement that is not in T, then x is blocked at the same point in both programs on that path. Assume x is bocked by a statement y that precedes it in $TS_1(P,x)$. By assumption, y is scheduled at least as early on all paths in $CG_1(P,x)$. The theorem follows from the case analysis of Theorem 5.1.4. □

Theorem 5.3.1 shows that compact-global is better than trace scheduling by trace scheduling standards. In general, it is not the case that $TS(P) \leq_p CG(P)$.

The two algorithms may produce incomparable programs, because compact-global's aggressiveness in moving statements on important paths may prevent some transformations on a less important path that trace scheduling performs.

One detail of compact-global deserves explanation. When compact-global is applied to a program, a node n may be copied before it is selected by the trace picker. This happens because copies of statements from the selected path may move on other paths as well. If a node is copied before the trace picker has selected it, the copied nodes are added to a list. When the node (or one of its copies) is finally selected by the trace picker, migrate is applied to the set of statements in all the duplicated nodes. Because nodes duplicated by the core transformations cannot be on the same path, migrate works correctly when applied to this set of statements.

The trace scheduling model of computation is weaker than our model. Trace scheduling permits only vines (or chains) of tests in a node, while we permit dags of tests. However, this is not why compact-global performs better than trace scheduling. Consider again the example of Figure 5.1. Assume that only the two assignments can move; therefore, the generality of the multi-way jump mechanism is not an issue. Compact-global moves the two statements to node A, as shown in Figure 5.10, regardless of the order in which paths are compacted.[7] Assuming that the trace picker repeatedly selects the leftmost remaining path for compaction, the final result of trace scheduling the program is shown in Figure 5.11. Trace scheduling produces lower quality (and in general slower) code than compact-global.

[7]Two copies of node F are created during compaction using compact-global; these are merged in Figure 5.10.

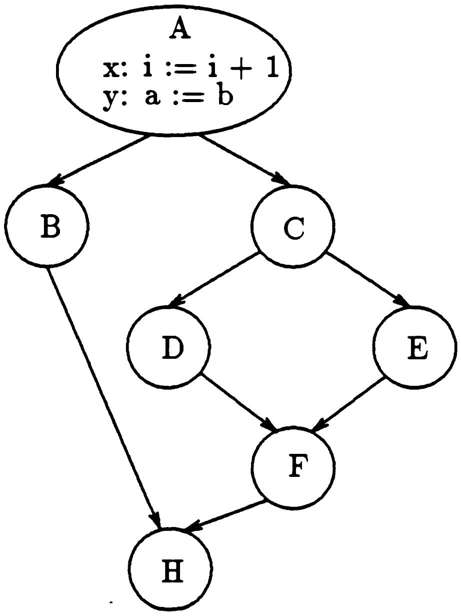


Figure 5.10: Final code after compact-global.

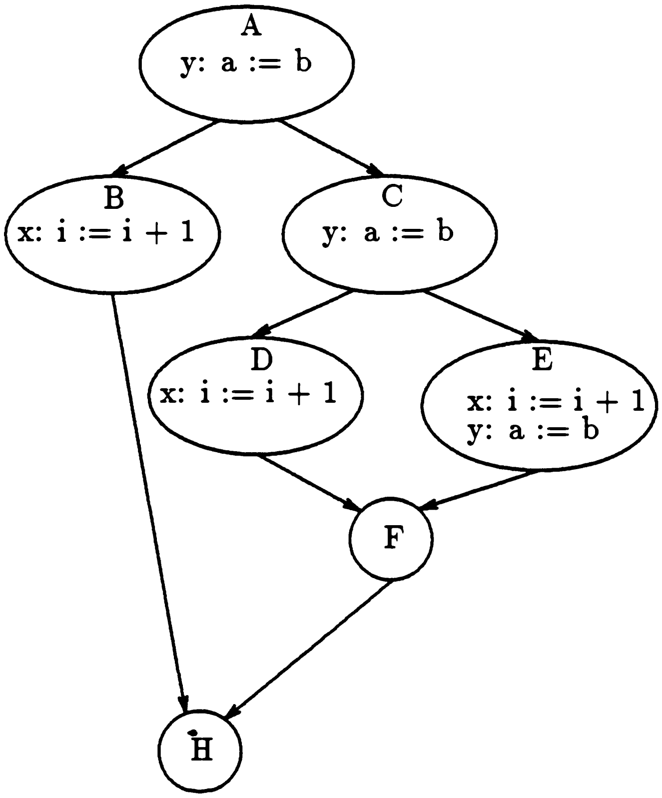


Figure 5.11: Final code after trace scheduling.

Chapter 6

Perfect Pipelining

In compaction-based parallelization, the standard method for extracting parallelism from a loop is to compact the loop body. This yields some performance improvement, but does not exploit parallelism that may be present between separate iterations of a loop. To extract more parallelism, most systems unroll the loop body a number of times before compacting. If a loop is unrolled k times, parallelism can be exploited inside this unrolled loop body, but the new loop still imposes sequentiality between every group of k iterations. In this chapter, we develop a parallelization technique, *perfect pipelining*, that overcomes this problem by achieving the effect of unbounded unrolling and compaction of a loop.

The loop shown in Figure 6.1 illustrates the importance of perfect pipelining. (The loop control code is simplified for clarity: the induction variable i is incremented implicitly on the backedge, as in a Fortran DO loop.) The running time of this loop is $4n$ steps, where n is the number of iterations executed. Multiple iterations of this loop may be overlapped, subject to the constraint that the first statement of an iteration is dependent on the result of the first statement of the previous iteration. Figure 6.2 shows a schedule after the loop has been unrolled three times and compacted.[1] Statement labels have been substituted

[1]Two additional memory locations are allocated to each array to handle the extra references generated when $i = n$. Alternatively, the loop bounds can be adjusted and a prologue added as

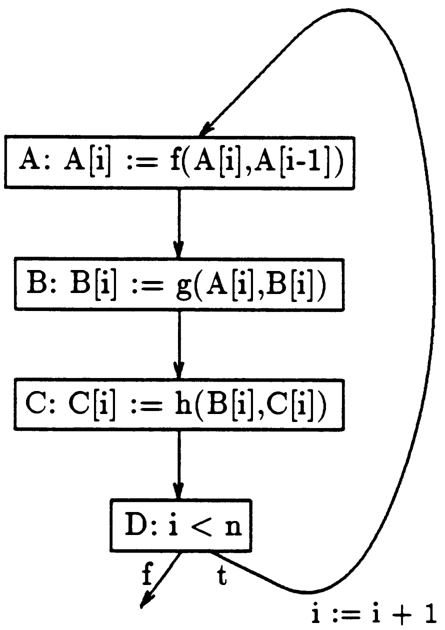


Figure 6.1: A sample loop.

for the statements; subscripts indicate the increment to the induction variable. The running time of this loop is $2n$ steps. Figure 6.3 shows the loop unrolled five times and compacted; in this case the running time is $\frac{8}{5}n$ steps. Note the low parallelism at the beginning and end of the loop body in both of these examples.

Additional unrolling and compaction improves the running time of this loop, but this approach becomes computationally expensive very rapidly. Perfect pipelining derives the program shown in Figure 6.4. Intuitively, perfect pipelining observes that the fourth and fifth nodes of Figure 6.3 execute the same statements from different iterations, and that further unrolling and compaction creates more nodes of the same type. We refer to this, informally, as the *pattern* generated by compaction of the loop.

The loop in Figure 6.4 is equivalent to an "infinitely unrolled" and compacted version of the original loop. The transformation achieves continuous (or perfect) pipelining of the loop iterations. The prologue and epilogue are everything not

---

in [CCK87].

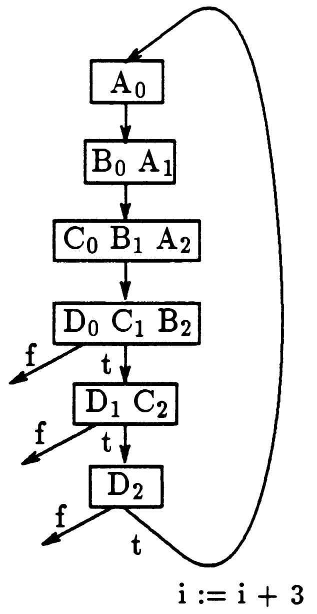


Figure 6.2: Loop unrolled three times and compacted

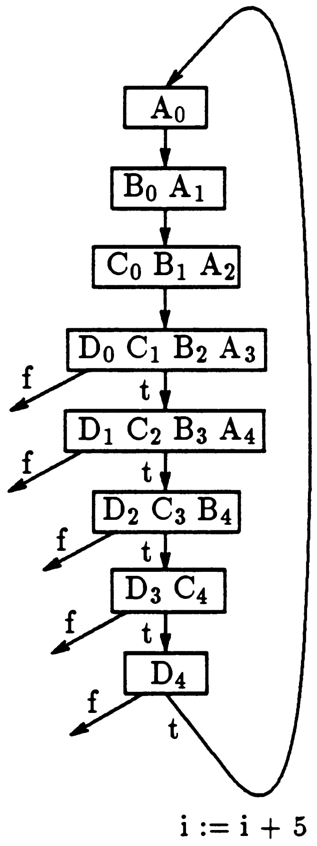
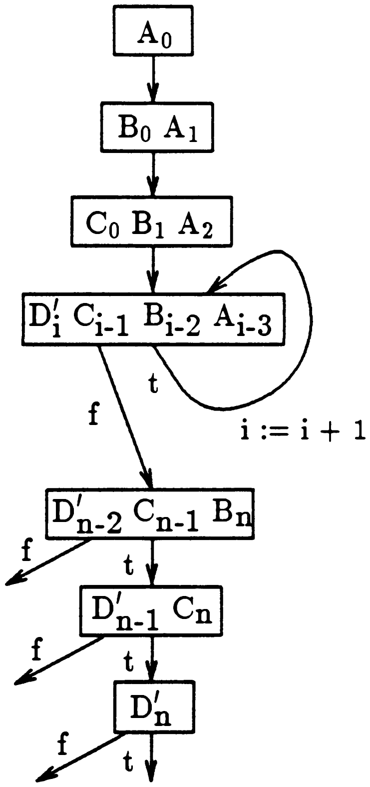


Figure 6.3: Loop unrolled five times and compacted.



where D': $i < n\text{-}2$

Figure 6.4: Loop after perfect pipelining.

included in the pattern. For Fortran-like DO loops, it turns out that the transformation requires little more than rearranging edges between nodes. (Note that the tests in the epilogue are unnecessary and can be deleted; we have left them in to emphasize the relationship between the loops before and after application of perfect pipelining.) The running time of the loop in Figure 6.4 is $n + 3$ steps.

In this example, the pattern detected by perfect pipelining is very simple because there are no branches (other than exits) in the loop body. Perfect pipelining finds such a pattern on all paths regardless of the flow of control within the loop body. For this reason, perfect pipelining performs well on loops where the flow of control is unpredictable.

Perfect pipelining applies in the presence of resource constraints. While it is infeasible to compute optimal loop schedules with resource constraints (even in idealized models [GJ79]), the fact that perfect pipelining achieves continuous overlap of iterations on all paths tends to keep machine utilization high. In this chapter, we prove that the transformation finds a pattern given fixed resources. In Chapter 7, we provide an example illustrating its performance when the loop has unpredictable flow of control and machine resources are a limiting factor.

Perfect pipelining can be combined with transformations that alter the dependencies of a loop. For example, indirect references are renamed in Figures 6.2 and 6.3 to eliminate dependencies caused by the induction variable. The use of transformations that change the dependencies of a loop require extensions that are discussed in Chapter 11.

Finally, perfect pipelining bounds the amount of unrolling required. In existing parallelization systems, particularly trace scheduling [Fis79,Fis81], there is a trade-off between loop unrolling and exploitable parallelism. As illustrated in Figure 6.3, additional unrolling and compaction of a loop body theoretically improves execution time. However, unrolling and compaction increase the size of a program, and machines have storage limitations. In particular, high-performance parallel machines use an instruction cache, and a loop too large for the cache will perform very poorly. Thus, existing systems unroll as much as possible, subject to hardware constraints. In contrast, when a pattern is detected by perfect pipelining, no further unrolling is required. While this pattern may still be too large for an instruction cache, perfect pipelining typically requires much less unrolling and yields very concise schedules. For example, to achieve performance within 10% of the loop in Figure 6.4 using trace scheduling, the loop in Figure 6.1 must be unrolled 30 times. To achieve performance within 1% of the loop in Figure 6.4 using trace scheduling, the loop in Figure 6.1 must be unrolled 300 times.

Perfect pipelining resides outside of the basic transformational system discussed in Chapter 3; no sequence of core transformations can achieve the loop in

Figure 6.4 from the loop in Figure 6.1. We use $\leq_p$ to establish the relationship between the core transformations and perfect pipelining.

This chapter is divided into six sections. The first section discusses the conditions under which it is possible to infer a repeating pattern from a loop. The second section develops the perfect pipelining algorithm. The third section investigates a second characterization of perfect pipelining. The fourth section proves the correctness of perfect pipelining. The last two sections discuss extensions to the basic framework of the first section. This chapter is a high-level and theoretical view of perfect pipelining; concrete examples are presented in subsequent chapters.

6.1 Conditions for Perfect Pipelining

It is intuitively obvious that the loop in Figure 6.4 is the "right" pipelined version of the loop in Figure 6.3. However, it is not obvious how a compiler can safely detect this fact by examining the finite number of loop iterations exposed by unrolling. Consider the following analogy with rational and irrational numbers:

> Answer the following question: I have a number in mind; the first five digits of its decimal expansion are 0.99999. Is my number rational or irrational?

This question cannot be answered since there are both rational and irrational numbers that begin 0.99999. Without more information, no repeating behavior can be inferred.

Returning to loops, we see that perfect pipelining cannot be used with every compaction algorithm. Consider Figure 6.5.[2] This compaction algorithm overlaps the statments of unrolled iteration L_i if the ith digit of π is one. Unrolling and compacting a loop with this compaction algorithm cannot reveal any pattern

[2]Recall that u^iL is shorthand for loop L unrolled i times, and that u^iL consists of unrolled iterations $L_0, \ldots, L_{i-1}$.

let L be a loop to be compacted
let i be an unrolling
for each unrolled iteration L_j in u^iL **do**
    Compact iteration L_j if the jth digit of the
        binary expansion of π is one.

Figure 6.5: An irregular compaction algorithm.

because the compaction algorithm exhibits irrational behavior. After formalizing what perfect pipelining achieves, we characterize compaction algorithms that can be used with perfect pipelining.

6.1.1 Formalization

We make precise the informal notion that a loop transformed by perfect pipelining is an infinitely unrolled and compacted version of the original loop. We assume that a program is a simple (innermost) loop; nested loops are treated in Chapter 9. Let $\mathcal{C}u^iL$ stand for a compaction algorithm $\mathcal{C}$ applied to u^iL. We give a method, for the class of compaction algorithms discussed below, to compute a program $\mathcal{C}u^\infty L$ satisfying

$$\forall i \, \mathcal{C}u^iL \leq_p \mathcal{C}u^\infty L$$

6.1.2 The Compaction Algorithm

The are two basic requirements that any compaction algorithm used with perfect pipelining must satisfy. First, $\mathcal{C}u^{i+1}L$ must be a better approximation to the final (pipelined) loop than $\mathcal{C}u^iL$; i.e., the compaction algorithm must be monotonic. Second, the compaction algorithm must exhibit some repeating behavior. This section formalizes these two conditions.

Definition 6.1.1 (Well-behaved) Let $\mathcal{C}$ be a deterministic compaction algorithm built from move-op, move-test, and delete. Consider the sequence $\mathcal{C}u^0L$,

Cu^1L, Cu^2L, ... If $\forall i\ Cu^iL \leq_p Cu^{i+1}L$, then C is *well-behaved* on L and the Cu^iL form a *chain.* (A chain is a set of programs totally ordered by $\leq_p$.)

This is the monotonicity condition—the compaction algorithm should perform better with larger unrollings of the loop. For the moment, we require that a compaction algorithm be well-behaved. This requirement is actually stronger than necessary; we explain why and develop a better monotonicity condition in Section 6.5.

To formalize the second requirement, we must define equivalence of program nodes. Nodes n and n' are *equivalent* if, in a large enough unrolled and compacted program, starting computation at n in a store s is indistinguishable from starting computation at n' in store s. A compaction algorithm C can be pipelined if C generates finitely many distinct classes of equivalent nodes. This is the formalization of the repeating behavior required for perfect pipelining. The definition of equivalent nodes requires some notation to discuss the "same" node in u^iL and $u^{i+j}L$.

Definition 6.1.2 A node n in L and a node n' in L' are equal, written "n in $L = n'$ in L'", if all acyclic paths from the start node to n and n' are identical in both programs.

Definition 6.1.3 (Equivalent Nodes) Two program nodes n and n' in Cu^iL are *equivalent* if, for any integer k, there is an unrolling j such that in $Cu^{i+j}L$:

$$\begin{aligned} \langle n_1, s\rangle \rightarrow \langle n_2, s_2\rangle \rightarrow \ \ldots \ &\rightarrow \langle n_k, s_k\rangle = \\ \langle n'_1, s\rangle \rightarrow \langle n'_2, s'_2\rangle \rightarrow \ \ldots \ &\rightarrow \langle n'_k, s'_k\rangle \\ \text{where } n \text{ in } Cu^iL &= n_1 \text{ in } Cu^{i+j}L \wedge \\ n' \text{ in } Cu^iL &= n'_1 \text{ in } Cu^{i+j}L \end{aligned}$$

Unless otherwise stated, we assume that compaction algorithms are well-behaved and generate only finitely many classes of equivalent nodes for a given loop.

6.2 The Algorithm

In this section we present the perfect pipelining algorithm. The algorithm is developed without reference to a particular compaction strategy. The results of this section are applied to several different compaction algorithms in subsequent chapters.

If nodes n and n' can be identified as equivalent, then perfect pipelining may delete n' and add edges from *pred*(n') to n. For example, in Figure 6.3 the third and fourth nodes are equivalent: for a sufficiently large unrolling with compaction, computations beginning at the two nodes appear identical.

Lemma 6.2.1 below establishes an important property exhibited by well-behaved compaction algorithms when machine resources are limited. While resource limits can be arbitrarily complex, we simply impose a limit c on the number of statements that a node can contain. We assume that the core transformations are modified to move statements only if the resulting program graph does not violate the resource constraints.

Lemma 6.2.1 Let C be well-behaved on L. For every node n there is an unrolling i such that n is unchanged for any unrolling greater than i. That is,

$$\exists i \text{ s.t. } \forall j > i \;\; n \text{ in } Cu^iL = n \text{ in } Cu^jL$$

Proof: Because C is well-behaved, a statement x in $op(n)$ in Cu^iL can only be scheduled in n or a predecessor of n in Cu^jL. The lemma follows immediately from the fact that resources are bounded. □

The following theorem establishes a key fact about the compaction algorithms under consideration (those that are well-behaved and generate only finitely many distinct classes of equivalent nodes).

Theorem 6.2.2 (Convergence) For a sufficiently large unrolling i, on every path in Cu^iL there are two equivalent nodes n and n' where n and n' do not

```
let i be the number in Theorem 6.2.2
P := Cu^i L;
for each path p through P do
    begin
        let n and n' be equivalent nodes on p;
        Replace edges (m, n') by (m, n);
        Delete any unreachable nodes.
    end;
```

Figure 6.6: Perfect pipelining.

change with greater unrolling and compaction.

Proof: Because resources are bounded, all paths must grow longer with larger unrollings. Therefore, because there are only finitely many classes of equivalent nodes, for a large enough unrolling every path has at least two equivalent nodes. The second clause of the theorem is guaranteed by Lemma 6.2.1. □

The perfect pipelining transformation is given in Figure 6.6. The algorithm finds equivalent nodes n and n' in the compacted program graph, deletes n', and adds backedges from the predecessors of n' to n. For clarity, we have presented the simplest possible form of Theorem 6.2.2 and the perfect pipelining algorithm. A stronger statement actually holds: it is not necessary that the equivalent nodes be on the same path—it is sufficient to find equivalent nodes anywhere in the loop body. This is important in practice—exploiting this property speeds convergence of the algorithm.

Refer again to the loop in Figure 6.1. In Figure 6.3, the loop has been unrolled five times and compacted.[3]. The fourth and fifth nodes are equivalent. The transformation deletes the fifth node and adds an edge from the fourth node to itself with an induction variable increment of one. The result of applying perfect pipelining to this loop is shown in Figure 6.4.

[3]The compaction algorithm used in this example is presented in Section 7.2.

6.3 More on Convergence

By Theorem 6.2.2, a sufficient condition for convergence of the perfect pipelining algorithm is that the compaction algorithm generates finitely many classes of equivalent nodes. This is also a necessary condition: if a compaction algorithm C generates infinitely many classes of equivalent nodes, then then it cannot be that there are two equivalent nodes on every path in any Cu^iL. In other words, if the compaction algorithm generates infinitely many classes of equivalent nodes, then there is no finite representation of $Cu^\infty L$. Thus, a well-behaved compaction strategy converges if and only if it generates finitely many classes of equivalent nodes.

Checking that a compaction algorithm generates only finitely many classes of equivalent nodes can be difficult. In this section, we study conditions that are easy to verify and that provide some insight into the important properties of those compaction algorithms that can be pipelined.

A necessary condition for convergence is a *bounded* compaction algorithm. Intuitively, when a loop is compacted with a bounded compaction algorithm, every statement in the compacted loop is executed at the same rate regardless of the unrolling chosen. In the following definition, let the *distance* between two nodes n and n' to be the length of the longest path from n to n' that does not include a backedge.

Definition 6.3.1 (Boundedness) Let x_i denote the statement x in unrolled iteration L_i. A compaction algorithm is *bounded* for loop L if for every integer j there is a constant k such that in Cu^hL, the distance between $node(x_i)$ and $node(y_{i+j})$ is at most k.

Consider the loop in Figure 6.7. The same loop unrolled five times and compacted using compact-global (see Chapter 5) is shown in Figure 6.8. We assume that up to six statements can be executed in parallel. Compact-global is

```
for i := 1 to n do
    begin
        A: A[i] := C[i];
        B: B[i] := B[i − 1]
    end;
```

Figure 6.7: Another sample loop.

not a bounded compaction algorithm for this loop; note that the first statement vectorizes while the second is involved in a recurrence. If resource constraints allow more than two statements to be executed in parallel, then the distance between statements A_i and B_i grows arbitrarily large as the loop is unrolled and compacted. In terms of Definition 6.1.3, compact-global generates an unbounded number of classes of equivalent nodes. In this example, for any unrolling and compaction using compact-global, every node is equivalent only to itself.

The fact that unbounded compaction algorithms cannot be pipelined is discouraging. It shows, for instance, that trace scheduling, or any algorithm that performs maximal compaction on at least one path, cannot be pipelined for all loops. However, this does not imply that there are no good pipelining strategies. The loop in Figure 6.9 has the same running time as the loop in Figure 6.8 and satisfies the boundedness condition. In Chapter 8, we develop a compaction algorithm that computes this schedule.

A bounded compaction algorithm is a necessary—but not sufficient—condition for convergence of perfect pipelining. The compaction algorithm in Figure 6.5 is bounded, but it also generates an unbounded number of classes of equivalent nodes. This compaction algorithm uses global information about the loop (the number of the iteration being compacted) to prevent any pattern from emerging. Thus, a compaction algorithm that converges should use only local information. For example, the compacted form of an iteration should depend only on a finite number of adjacent iterations. A local compaction algorithm cannot use global

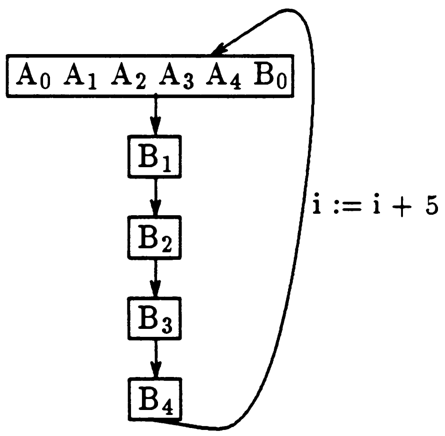


Figure 6.8: The loop unrolled and compacted with compact-global.

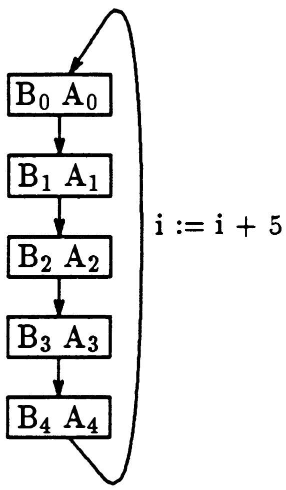


Figure 6.9: The loop unrolled and compacted using a bounded algorithm.

analysis of the loop to decide where to move a given statement. The ideas of locality and boundedness in compaction algorithms are formalized by the following theorem.

Definition 6.3.2 A *finite automaton* M is a machine with a one-way read-only input tape, a one-way write-only output tape, and a finite internal state, where the size of the state is dependent only on L. At each step, M may read one symbol from the input tape, write one symbol on the output tape, and make an internal state transition.[4]

Theorem 6.3.3 Perfect pipelining converges with compaction algorithm $\mathcal{C}$ if, for a given loop L, $\mathcal{C}$ is well-behaved and $\mathcal{C}$ acts as a finite automaton.

Proof: Because $\mathcal{C}$ has a finite control, it has at most a finite number of distinct states. A classical result of automata theory shows that a finite-state machine must loop (repeat states) on sufficiently large inputs [HU79]. Thus, a compaction algorithm with a finite state must produce equivalent nodes on all paths for a sufficiently large unrolling. □

Locality limits the size of the state. In Figure 6.5, the compaction algorithm computes π to arbitrary precision, so it is not a finite-state compaction algorithm. An unbounded compaction algorithm also has unbounded state: no finite-state machine can generate more than a finite number of distinct classes of equivalent nodes. There are, of course, compaction algorithms with unbounded state which converge. Thus, Theorem 6.3.3 provides a sufficient, but not necessary, condition for convergence.

6.4 Correctness of Perfect Pipelining

We prove that perfect pipelining is correct and that it is equivalent to infinite unrolling and compaction using a compaction algorithm $\mathcal{C}$. The following lemma

[4] This is actually a special variety of finite automaton, the Mealy machine [HU79].

simplifies the proof. Recall that $T(P, s)$ is the execution history of program P in initial store s (Definition 2.2.1).

Lemma 6.4.1 Let $\mathcal{C}u^{\infty}L$ be the result of perfect pipelining. For any integer k, there is an unrolling i such that $T(\mathcal{C}u^{\infty}L, s)$ is identical to $T(\mathcal{C}u^{i}L, s)$ for the first k steps.

Proof: Choose an i such that every node on a path of length k or less from the start node in $\mathcal{C}u^{i}L$ is unchanged with larger unrollings and compaction. (The existence of i is guaranteed by Lemma 6.2.1.) We proceed by induction on the number of times $T(\mathcal{C}u^{\infty}L, s)$ crosses one of the edges introduced by the perfect pipelining algorithm.

Basis: The trace $T(\mathcal{C}u^{\infty}L, s)$ crosses no edges introduced by the algorithm. By Lemma 6.2.1 and the choice of unrolling in the algorithm, the traces are identical for k steps.

Step: Assume that $\langle n_0, s\rangle \xrightarrow{*} \langle n_j, s_j\rangle$ in both programs. Let $\langle n_j, s_j\rangle \rightarrow \langle n_{j+1}, s_{j+1}\rangle$ in $\mathcal{C}u^{i}L$, and let $\langle n_j, s_j\rangle \rightarrow \langle n, s_{j+1}\rangle$ in $\mathcal{C}u^{\infty}L$, where (n_j, n) is an edge introduced by the algorithm. The algorithm replaced edge (n_j, n') with (n_j, n), where n' and n are equivalent nodes. By construction, n' is equivalent to some node n_h executed earlier in $T(\mathcal{C}u^{\infty}L, s)$. This follows because there are equivalent nodes on every path in the loop that is input to the algorithm. By assumption, $\langle n_0, s\rangle \xrightarrow{*} \langle n_h, s_h\rangle$ in both traces. But n_h in $T(\mathcal{C}u^{i}L, s)$ is equivalent to n_{j+1}, implying that n and n_{j+1} equal. Furthermore, by the definition of equivalent nodes, everything executed up to the next edge introduced by the algorithm (or up to n_k, whichever occurs first) must be the same in both traces. $\square$

Theorem 6.4.2 For all i and L for which $\mathcal{C}$ is well-behaved and generates only finitely many classes of equivalent nodes:

$$\mathcal{C}u^{i}L \leq_p \mathcal{C}u^{\infty}L$$

Proof: Let k be the length of $T(\mathcal{C}u^iL, s)$. By Lemma 6.4.1, there is a program $\mathcal{C}u^jL$, where $j \gg \max(i, k)$, such that $T(\mathcal{C}u^jL, s) = T(\mathcal{C}u^\infty L, s)$. Because $\mathcal{C}$ is well-behaved, $\mathcal{C}u^iL \leq_p \mathcal{C}u^jL$. We conclude that $\mathcal{C}u^iL \leq_p \mathcal{C}u^\infty L$. □

Theorem 6.4.2 shows that perfect pipelining is as good as full unrolling and compaction on all paths. The transformation computes a closed form of the pattern generated by repeated unrolling and compaction using $\mathcal{C}$.

6.5 More on Monotonicity

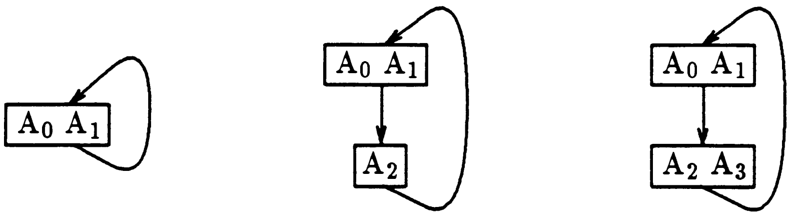
As mentioned in Section 6.1.2, the original definition of a well-behaved compaction algorithm is too strong. While the results of the previous sections are true for any well-behaved compaction algorithm, there are interesting compaction algorithms that converge but are not well-behaved. Consider Figure 6.10. We assume that at most two statements can be executed in parallel due to resource constraints. We use compact-global ($\mathcal{G}$) as the compaction algorithm for this example. Because the loop body has only a single statement and no dependencies, compact-global is a bounded and local compaction algorithm for this loop. Figure 6.10b shows $\mathcal{G}u^2L$, $\mathcal{G}u^3L$, and $\mathcal{G}u^4L$. Note that $\mathcal{G}u^2L \not\leq_p \mathcal{G}u^3L$ and $\mathcal{G}u^2L \equiv_p \mathcal{G}u^4L \equiv_p \mathcal{G}u^\infty L$. Perfect pipelining converges with $\mathcal{G}u^4L$ because the two nodes are equivalent.

When resources are a limiting factor, a loop unrolled and compacted i times may be faster than a loop unrolled and compacted $i+j$ times because the smaller loop fits the resources better. (Note that this was not relevant in Chapter 5 because we did not consider resource-constrained scheduling.) The purpose of this section is to extend our theory to cover all interesting compaction algorithms not satisfying the constraints of previous sections. To do this, we weaken the definition of a well-behaved compaction algorithm.

Definition 6.5.1 $\mathcal{C}$ is *well-behaved* if $\forall i, k \;\; \mathcal{C}u^iL \leq_p \mathcal{C}u^{2i}L$.

```
for i := 1 to N do
    A: A[i] := A[i] + 1;
```

(a) A simple loop.



$\mathcal{G}u^2L$ $\qquad$ $\mathcal{G}u^3L$ $\qquad$ $\mathcal{G}u^4L$

(b) The loop unrolled 2, 3, and 4 times and compacted.

Figure 6.10: An example that is not well-behaved.

Definition 6.5.1 only requires that a $\mathcal{C}u^{2i}L$ be at least as good as duplicating the loop body of $\mathcal{C}u^{i}L$, so this is quite a weak constraint. The definition of equivalent nodes must be changed to reflect the change in the definition of well-behaved.

Definition 6.5.2 Two program nodes n and n' in $\mathcal{C}u^{i}L$ are *equivalent* if, for any integer k, there is an integer $j = 2^c$ such that in $\mathcal{C}u^{ij}L$:

$$\begin{aligned}
\langle n_1, s_1\rangle \rightarrow \langle n_2, s_2\rangle \rightarrow \quad \ldots \quad &\rightarrow \langle n_k, s_k\rangle = \\
\langle n'_1, s_1\rangle \rightarrow \langle n'_2, s'_2\rangle \rightarrow \quad \ldots \quad &\rightarrow \langle n'_k, s'_k\rangle \text{ where} \\
n \text{ in } \mathcal{C}u^{i}L \quad &= \quad n_1 \text{ in } \mathcal{C}u^{ij}L \wedge \\
n' \text{ in } \mathcal{C}u^{i}L \quad &= \quad n'_1 \text{ in } \mathcal{C}u^{ij}L
\end{aligned}$$

All of the results of the previous sections hold using Definitions 6.5.1 and 6.5.2; for any chain $\mathcal{C}u^{i_0}L, \mathcal{C}u^{i_1}L, \ldots$ where $\mathcal{C}u^{i_j}L \leq_p \mathcal{C}u^{i_{j+1}}L$, perfect pipelining computes a loop $\mathcal{C}u^{\infty}_{i}L$ such that $\forall j\ \mathcal{C}u^{i_j}L \leq_p \mathcal{C}u^{\infty}_{i}L$. However, because the definition of well-behaved is so weak, the result of perfect pipelining using different chains may be different; $\mathcal{C}u^{\infty}L$ may not be unique. Let the *canonical chain* be $\mathcal{C}u^{0}L, \mathcal{C}u^{1}L, \mathcal{C}u^{2}L, \mathcal{C}u^{4}L, \mathcal{C}u^{8}L, \ldots$. We define $\mathcal{C}u^{\infty}L$ to be the result of perfect pipelining using the canonical chain.

Using this extended definition of well-behaved, our system can express a limited form of vectorization. Consider again Figure 6.10. The original loop in this example consists of a single statement with no data dependencies. The loop $\mathcal{G}u^{\infty}L$ executes k statements at a time, where k is determined by the resource constraints of the machine. This is also what vectorization achieves on this example. Thus, vectorization of a single statement can be viewed as a limit of an infinite sequence of core transformations.

6.6 Perfect Pipelining and Dependency Analysis

In Section 1.4, we state that the power of compaction is relative to the power of the underlying dependency analysis. We formalize this statement and investigate the impact of dependency analysis on perfect pipelining. We show that perfect pipelining can achieve the effect of compaction with arbitrarily precise dependency information.

6.6.1 Dependency Analyzers

A *dependency analyzer* is a procedure D that takes two program statements and returns $\top$ if there is no dependency between the statements and $\perp$ if there is a dependency. More formally, there is a dependency between statements X and Y if there is a trace $T(P, s)$ such that executed occurrences (*instances*) x of X and y of Y satisfy $x \prec y$ in $T(P, s)$ (Definition 2.3.1). Let X be the set of all instances of a program statement x in all execution traces of a program L.

To make the notation of this section clear, we identify a program statement with its set of instances. Thus, an upper-case letter is used for a program statement and its set of instances, while a lower-case letter is used for a particular instance. To allow comparison of different dependency analyzers for a loop, we define dependency analysis on the instances of statements rather than on the statements themselves. Finally, to allow us to distinguish runtime dependency information from static (approximate) dependency information, we add the initial store of a computation as an argument to dependency analysis.

We first define *perfect* dependency analysis; this recasts Definition 2.3.1 in the context of a dependency analyzer. For simplicity, we consider only true dependencies.

Definition 6.6.1

$$D^{\mathrm{perf}}(x,y,s) = \begin{cases} \bot & x \prec y \text{ in } T(L,s) \\ \top & \text{otherwise} \end{cases}$$

The dependency analyzer D^{perf} represents perfect runtime dependency information. The following definition describes approximate dependency analyzers. Each D^i captures the most precise information that can be obtained by examining the first i iterations of a loop.

Definition 6.6.2

$$D^i(x,y,s) = \begin{cases} \bot & \text{if } \exists s' \text{ s.t. } x' \prec y' \text{ in } T(u^i L, s') \wedge x', x \in X,\ y', y \in Y \\ \top & \text{otherwise} \end{cases}$$

Note that the third argument to D^i (the initial store) has no effect on the result. Statically computed information cannot, in general, predict the state of a computation at runtime. Again, we have included the initial store as an argument to permit the definition of D^{perf}. For clarity, we omit the store in what follows.

Let $\top$ and $\bot$ be a two-element domain ordered in the usual way (i.e., $\bot \leq \top$). All loops $u^i L$ have the same execution traces, so it is meaningful to compare the different dependency analyzers of Definition 6.6.2. We define a partial order on dependency analyzers in the usual way:

$$D \leq D' \Leftrightarrow D(x,y) \leq D'(x,y)$$

If $D \leq D'$, then D' is more precise—reports fewer dependencies—than D.

Lemma 6.6.3 The D^i are *conservative:*

$$\forall i \; D^i \leq D^{\text{perf}}$$

Proof: By Definition 6.6.2, $\exists s$ s.t. $D^{\text{perf}}(x,y) = \perp \Rightarrow D^i(x,y) = \perp$. □

We are interested only in conservative dependency analysis; if compaction is performed using a dependency procedure that is not conservative, the compacted program may not preserve the semantics of the original program. The following lemma shows that dependency analysis increases in accuracy when more loop iterations are examined.

Lemma 6.6.4 $D^i \leq D^{ki}$.

Proof: Each X in u^iL is the union of k sets X_1 through X_k in $u^{ki}L$, therefore $D^i \leq D^{ki}$. □

Consider the domain of all conservative dependency analyzers. This domain is a complete lattice with D^{perf} as a top element (a proof of this fact is included in the appendix). We define D^∞ to be the least upper bound of the D^i of the canonical chain:

$$D^\infty = \bigsqcup_i D^i \text{ where } i = 2^c$$

Because the domain is a complete lattice, D^∞ exists [Sto77].

By the standards of dependency analysis in the literature, our definition of the D^i is simplistic. The D^i do not, for example, distinguish whether a dependency is loop-carried or loop-independent.[5] The definition can be extended to distinguish different types of dependencies, but this added complexity is unnecessary. The dependency information of D^∞ is the best a compiler could compute: it is at least as accurate as the information obtainable from examining any finite number of iterations.

[5]Recall that a loop-carried dependency is between statements in different iterations; a loop-independent dependency is between statements of the same iteration.

Note that D^{∞} is not necessarily equal to D^{perf}. In general, D^{∞} is less precise than D^{perf} because D^{∞} is conservative over all possible execution traces. We show that with perfect pipelining it is possible to achieve the effect of compaction using D^{∞}.

We extend our notation to include dependency analysis as a parameter to a compaction algorithm. Let $\mathcal{C}(u^iL, D^j)$ stand for loop L unrolled i times and compacted using $\mathcal{C}$ with dependency analyzer D^j. In compacting u^iL using D^j, there is a dependency between statements X and Y of u^iL if there is an $x \in X$ and $y \in Y$ such that $D^j(x,y) = \perp$. We also extend the definition of a well-behaved compaction algorithm to include dependency analysis.

Definition 6.6.5 A compaction algorithm $\mathcal{C}$ is *well-behaved* on L if:

- $\mathcal{C}(u^iL, D^j) \leq_p \mathcal{C}(u^{2i}L, D^j)$
- $D^j \leq D^k \Rightarrow \mathcal{C}(u^iL, D^j) \leq_p \mathcal{C}(u^iL, D^k)$

The first point is the natural extension of Definition 6.5.1. The second point says that a well-behaved compaction algorithm should do better with better dependency analysis. The compaction algorithms in which we are interested are well-behaved by Definition 6.6.5. We also require that a compaction algorithm generate only finitely many classes of equivalent nodes. We extend this constraint to include dependency analysis.

Constraint 6.6.6 For every L there is a constant c such that $\mathcal{C}(u^iL, D^j)$ has at most c classes of equivalent nodes.

Unless otherwise stated, we assume that a compaction algorithm is well-behaved and generates finitely many classes of equivalent nodes.

6.6.2 Pipelining with Dependency Analyzers

We first show that perfect pipelining produces better results with better dependency analysis.

Lemma 6.6.7 Let $L_1^1 \leq_p L_1^2 \leq_p \ldots \leq_p L_1^\infty$ and $L_2^1 \leq_p L_2^2 \leq_p \ldots \leq_p L_2^\infty$ be chains with pipelined limits L_1^∞ and L_2^∞ respectively. Then

$$\forall i \exists j \text{ s.t. } L_1^i \leq_p L_2^j \Rightarrow L_1^\infty \leq_p L_2^\infty$$

Proof: By Theorem 6.4.1, for any computation $T(L_1^\infty, s)$ there is an i such that $T(L_1^\infty, s) = T(L_1^i, s)$. By assumption and by Theorem 6.4.2, there is a j such that $L_1^i \leq_p L_2^j \leq_p L_2^\infty$. Therefore, $L_1^\infty \leq_p L_2^\infty$. □

Theorem 6.6.8 Consider two chains $\mathcal{C}(u^i L, D^j), \mathcal{C}(u^{2i} L, D^j), \ldots$ and $\mathcal{C}(u^i L, D^k)$, $\mathcal{C}(u^{2i} L, D^k), \ldots$. If $D^j \leq D^k$, then $\mathcal{C}(u_i^\infty, D^j) \leq \mathcal{C}(u_i^\infty, D^k)$.

Proof: By Definition 6.6.5, $D^j \leq D^k \Rightarrow \mathcal{C}(u^i L, D^j) \leq_p \mathcal{C}(u^i L, D^k)$. The theorem follows by Lemma 6.6.7. □

As one would hope, simply substituting a better dependency analyzer improves perfect pipelining. However, a better result is possible. Consider the sequence $\mathcal{C}(u^1 L, D^1), \mathcal{C}(u^2 L, D^2), \mathcal{C}(u^4 L, D^4), \ldots$. By Definition 6.6.5, $\mathcal{C}(u^i L, D^i) \leq_p \mathcal{C}(u^{2i} L, D^i) \leq_p \mathcal{C}(u^{2i} L, D^{2i})$. Therefore, this is a chain. By Constraint 6.6.6 and Theorem 6.2.2, perfect pipelining with this chain computes a loop L' satisfying

$$\forall i \text{ where } i = 2^c,\ \mathcal{C}(u^i L, D^i) \leq_p L'$$

We show that L' is as good as perfect pipelining using D^∞. The following lemma allows us to prove that for any $j = 2^c$, $\mathcal{C}(u^j L, D^\infty) \leq_p L'$.

Lemma 6.6.9 For all j there exists a k, where $k = 2^d$, such that

$$\mathcal{C}(u^j L, D^\infty) = \mathcal{C}(u^j L, D^k)$$

Proof: There are only finitely many parallelizations of $u^j L$ achievable using move-op, move-test, and delete. The lemma follows from the fact that $\mathcal{C}$ is well-behaved. (This lemma holds with or without resource constraints.) □

Theorem 6.6.10 For all j where $j = 2^c$ the following holds:

$$\begin{aligned} \mathcal{C}(u^j L, D^\infty) &\leq_p L' \wedge \\ \mathcal{C}(u^\infty L, D^j) &\leq_p L' \end{aligned}$$

Proof: Consider the first conjunct. By Lemma 6.6.9, there is a k, where $k = 2^d$, such that $\mathcal{C}(u^j L, D^\infty) \equiv_p \mathcal{C}(u^j L, D^k)$. Then, $\mathcal{C}(u^j L, D^k) \leq_p \mathcal{C}(u^{jk} L, D^{jk}) \leq_p L'$. Consider the second conjunct. Let $i = 2^d$. Clearly, $\forall i\ \mathcal{C}(u^i L, D^j) \leq_p \mathcal{C}(u^{ij} L, D^{ij})$; the result follows from Lemma 6.6.7. □

Theorem 6.6.10 shows that perfect pipelining can achieve the effect of full unrolling and compaction with arbitrarily precise dependency information using only partial unrolling and approximate dependency information. Given the result of Theorem 6.6.10, we write $\mathcal{C}(u^\infty L, D^\infty)$ for L'.

6.7 Discussion

This chapter focuses on the general properties of perfect pipelining. An important issue that we have not addressed is how perfect pipelining detects equivalent nodes. The procedure for detecting equivalent nodes is very dependent on the compaction algorithm used. It is easy to show that the problem of detecting equivalent nodes is undecidable for an arbitrary compaction algorithm; thus, just because a compaction algorithm is guaranteed to converge does not imply that convergence can be detected. The problem of detecting equivalent nodes is decidable for any compaction algorithm satisfying the finite automata characterization of Theorem 6.3.3.

Detecting equivalent nodes can be undecidable for a given compaction algorithm if the dependencies of the loop are complex and the compaction algorithm exploits that complexity. All previous work in software pipelining (including the author's) makes strong assumptions about the dependencies of the loop to guarantee that detecting equivalent nodes is decidable [AN87b,SDX87,Ebc87,Lam87,

```
for i := 1 to N do
    A: A[i + 3] := A[2i];
```

Figure 6.11. A program with irregular dependencies.

AN88c]. In previous work, the dependency analysis is essentially fixed to some D^j for all loops (j is generally 1). For example, in [Ebc87] no iterations are allowed to completely overlap on any path; the first statement of iteration L_i must be scheduled after the first statement of L_{i-1}. This is a restriction on the compaction algorithm, but it has the same effect as assuming that there is a loop-carried dependency between the first statements of L_{i-1} and L_i.

Implicit in the same previous work is the assumption that software pipelining cannot be done with arbitrarily accurate dependency information. The results of Section 6.6 show that this is not necessarily the case; it is theoretically possible to achieve the effect of perfect pipelining using D^∞. However, the fact that undecidability can arise from dependency information appears to weaken Theorem 6.6.10. The existence of a loop which is as good as full unrolling and compaction with precise dependency information is uninteresting if it is uncomputable.

We present an example demonstrating that there are interesting cases where loop $\mathcal{C}(u^\infty L, D^\infty)$ is easily computed, and settling for $\mathcal{C}(u^\infty, D^1)$ is substantially worse. Our example is given in Figure 6.11. The dependencies are contrived so that the loop does not vectorize, and so that previous software pipelining techniques perform poorly. We do not know of any existing technique which can parallelize this loop, other than adding statements to test dependencies at runtime [Nic84,Pol87,Nic88]. Let $\mathcal{G}$ be compact-global. It is easy to show that compact-global is bounded and local for any loop with a single statement if machine resources are fixed and array subscripts are linear expressions of a single iteration variable. Thus, compact-global is bounded and local for this loop.

For clarity, the loop control code is omitted. We assume that up to three statements can be executed in parallel. The sequence of loops computed be-

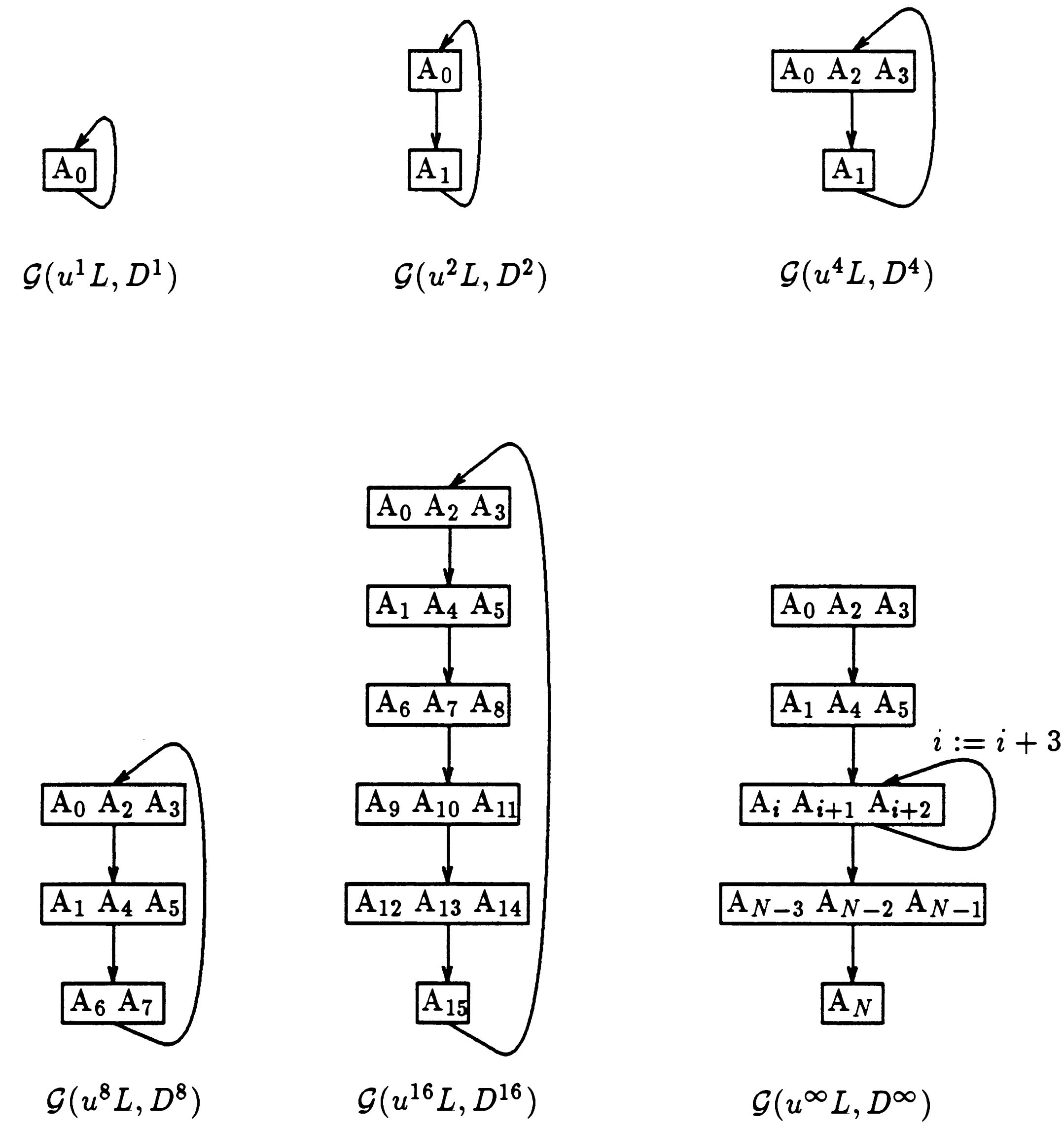


Figure 6.12: The chain of approximations to $\mathcal{G}(u^\infty L, D^\infty)$.

fore convergence is shown in Figure 6.12. The third, fourth, and fifth nodes of $\mathcal{G}(u^{16}L, D^{16})$ are equivalent. The details of how the pipelining algorithm detects that these nodes are equivalent requires concepts developed in Section A.1. The pipelining algorithm adds a backedge from the third node to itself and deletes the fourth node.

Suppose that the dependency analysis is restricted to D^1. Dependency analyzer D^1 assumes that there is a dependency between all iterations if there is a dependency between any iterations. Because there is a dependency between the first two iterations (the first iteration writes $A[4]$ and the second iteration reads $A[4]$), the pipelined loop $\mathcal{G}(u^\infty, D^1)$ is just the original loop body. Similar examples can be constructed for this algorithm showing that $\mathcal{G}(u^\infty L, D^j)$ is worse than $\mathcal{G}(u^\infty, D^\infty)$ for any j.

We do not have a general algorithm for perfect pipelining using D^∞; this is a topic of current research. The first step is to characterize the dependencies for which it is possible to pipeline with arbitrary analysis, without encountering computability problems. We conjecture that for loops with the very regular dependency structures encountered in practice, it is possible to achieve the effect of perfect pipelining using D^∞.

Chapter 7

Perfect Pipelining: A Case Study

In this chapter, we develop one compaction algorithm for use with perfect pipelining, the *simple rule*. We use this algorithm to illustrate how perfect pipelining applies to loops with arbitrary flow of control. We compare the effect of perfect pipelining using the simple rule with a standard multiprocessor parallelization technique, *doacross* [Cyt84].

7.1 The Dependencies

Recall that a loop-carried dependency [All83] is a dependency between separate iterations of a loop (see Section 1.4). In this context we are referring to the approximate dependencies a compiler computes using conservative analysis, rather than the precise trace dependency graphs used to define $\leq_p$ (Definition 2.3.1). We consider single (non-nested) loops satisfying the following constraint for any unrolling:

Constraint 7.1.1 Assume there is a dependency between statement x and y in L. If the dependency is loop-carried, then in u^iL there is a dependency between statement x of L_j and y of L_{j+1} for all j. If the dependency is loop-independent, then in u^iL there is a dependency between statement x of L_j and y of L_j for all j.

procedure *move*(x, n, m)
  if x is an assignment
    then $P :=$ *move-op*$(P, x,$ *move-set*$, m)$
      where *move-set* contains every member of *succ*(m) with
      a copy of x that can move
    else $P :=$ *move-test*$(P,$ *move-set*$, n, m)$
      where *move-set* contains all copies of x in *node*(x);

Figure 7.1: The move transformation.

Constraint 7.1.1 is equivalent to using dependency analyzer D^1, with the added ability to distinguish between loop-carried and loop-independent dependencies. Intuitively, the constraint requires that the dependencies of a loop unrolled i times are a good predictor of the dependencies of the loop unrolled $i + 1$ times. The constraint is quite restrictive; even so, many loops can be automatically rewritten to satisfy it (i.e., can be rewritten so that perfect pipelining using D^1 is equivalent to perfect pipelining using D^∞). Many loops arising in practice have dependencies with a *consistent threshold* [AK84]. The threshold [CCK87] of a dependency is the number of iterations it spans; a threshold is consistent if it is the same for all instances of a dependency. For example, the dependency caused by the statement $A[I + c] := A[I]$ in a loop with iteration variable I has a consistent threshold c. If a dependency has a consistent threshold c, then unrolling the the loop c times guarantees that loop-carried dependencies are between adjacent iterations. We develop an algorithm for performing this transformation automatically in Chapter 9.

7.2 The Simple Rule

To simplify the algorithms, we combine the primitives move-op and move-test into one transformation *move* (Figure 7.1). The simple rule moves an iteration L_j as far "up" in the program graph on as many paths as possible. The structure

procedure *move-iteration*(x, n, m)
  begin
    if $x \in op(n)$
      then
        begin
          move(x, n, m);
          if move failed **then fail**
        end;
    (* *next-op-in-it*(x, n, p) is next statement in the iteration
      after x on edge (n, p). *)
    for each $\langle p, y\rangle$ s.t. $p \in succ(n) \wedge$ *next-op-in-it*$(x, n, p) = y$ **do**
      move-iteration(y, p, n);
    Delete all empty nodes.
  end;

Figure 7.2: Moving an iteration.

of the simple rule is the same as migrate (Figure 5.5), but the unit of interest is a loop iteration instead of a statement. In the simple rule, statements in the iteration remain in adjacent nodes and the iteration keeps its "shape"—statements appear in the order of the original loop body.

The basic step of the simple rule is to move each statement in one copy of an iteration up one node in the program graph. An algorithm that accomplishes this is given in Figure 7.2. We assume that statements are identified with their unrolled iteration L_j. A **fail** command causes the entire recursive computation to terminate and restores the original program graph.

The simple rule is given in Figure 7.3. As in migrate, as many statements are combined as possible, thus minimizing code explosion. As iterations move through the program graph, copies of statements—forming distinct copies of the iteration—are generated where paths split. The top-level algorithm refers to the first statement in each copy of the iteration; the other statements are handled by move-iteration.

(* Let $P = u^i L$ *)
for each iteration $L_0, \ldots, L_i$ **do**
  begin
    $X := \{x\}$ where x is the first statement in L_j;
    repeat
      (* we assume that X always contains all copies of statement x *)
      Select $x \in X$ such that x's iteration can move and
        the depth of m in the graph is maximized;
      move-iteration$(x, node(x), m)$
    until no iteration can move;
    Delete all empty nodes.
  end;

Figure 7.3: The simple rule.

In the remainder of this chapter, we use $\mathcal{C}$ for the simple rule. An important property of $\mathcal{C}$ is that it is maximal—for any $\mathcal{C}'$ using move-iteration and for all programs P and unrollings i, $\mathcal{C}u^iP \not<_p \mathcal{C}'u^iP$. The simple rule is well-behaved, bounded, and local.

Figure 6.3 shows a loop unrolled and compacted using $\mathcal{C}$. The only loop-carried dependency is between the first statement of consecutive iterations; after application of $\mathcal{C}$ the iterations overlap, staggered by one node. The following lemma shows that detecting equivalent nodes is easy for the simple rule—nodes with the same statements from different iterations are equivalent.

Lemma 7.2.1 Let n and n' be nodes in $\mathcal{C}u^iL$, and let x_j be the occurrence of statement x in L_j. Assume that n and n' are unaffected by larger unrollings and applications of $\mathcal{C}$. If there is a k such that

$$x_j \in op(n) \Leftrightarrow x_{j+k} \in op(n')$$

and n and n' have the same dag structure, then n and n' are equivalent.

Proof: Assume i is large enough that the successors of n and n' are unaffected by larger unrollings and applications of $\mathcal{C}$. The existence of i is guaranteed by Lemma 6.2.1. Let p and p' be corresponding successors of n and n' respectively. Because statements from an iteration are adjacent and in the same order after the application of $\mathcal{C}$, it follows that x is a statement of iteration L_j in n and y is a statement in L_j in p if and only if x is a statement of L_{j+k} in n' and y is a statement of L_{j+k} in p'. The only other possibility is that p contains the first statement of an iteration L_j. If there is a loop-carried dependency, then, because loop-carried dependencies are regular and $\mathcal{C}$ is maximal, a moving iteration is always blocked by the same statement from the previous iteration on a particular path. Thus, the first statement of iteration L_j is in p if and only if the first statement of L_{j+k} is in p'. The fact that p and p' have the same dag structure follows from the definition of the simple rule and the fact that n and n' have the same dag structure. Applying this argument inductively to p and p', it follows that n and n' are equivalent nodes. □

7.3 An Example

Figure 7.4 shows a simple loop L. The loop searches an array of elements, saving the position of all elements that match a key in order on a separate list. As before, we have omitted the loop control code. There is also no exit test; we stress that this is only for simplicity. We assume that the target machine can execute up to three tests in parallel.

This particular loop highlights the problem that unpredictable flow of control presents in parallelization. Note that while the path consisting of all true branches has tight dependencies preventing speedup, the path consisting of all false branches has no dependencies whatsoever. Other paths (some true branches, some false branches) have intermediate parallelism.

Existing restructuring transformations for multiprocessors do little with such

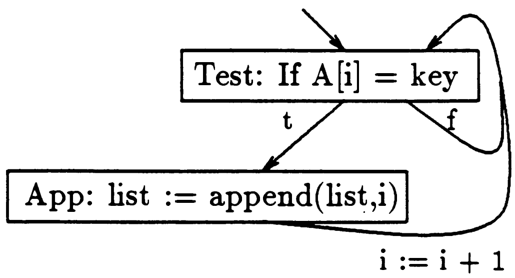
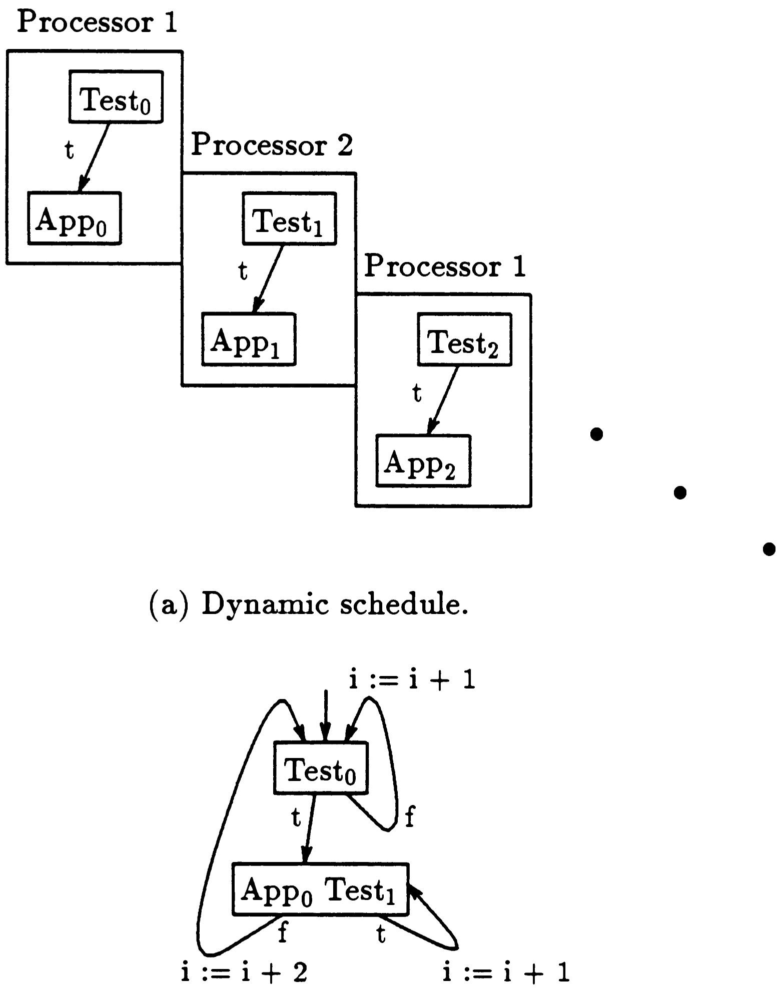


Figure 7.4: A simple loop L.

a loop. *Doacross* is a transformation that assigns the iterations of a loop to the processors of a synchronous or asynchronous multiprocessor [Cyt86]. Doacross computes a delay that must be observed between the start of a loop iteration L_i and the start of L_{i+1} on each path of L_i. For this loop, the computed delay is one on both paths; that is, iteration $i+1$ may begin after iteration i has executed its first statement. The dynamic execution of this loop using doacross is shown in Figure 7.5a. An equivalent static SPL schedule is shown in Figure 7.5b.

We now show how perfect pipelining with the simple rule applies to this loop. In Figure 7.6, the original loop is unrolled seven times. The statements have been replaced by labels with subscripts indicating the increment to the induction variable. The result of applying the simple rule is shown in Figure 7.7. The dag of tests within each node is arranged as a chain with the false branches pointing to the next test and the true branches exiting the node. (This is the structure of a lisp *cond* clause.) The lowest numbered test is the root of the dag.

The first four nodes in the left column of Figure 7.7 are equivalent and the start node is equivalent to the first two nodes in the right column. Figure 7.8 shows the result of applying perfect pipelining—only the first two nodes remain. In this pipelined loop, three tests are executed at each step. If T_j is the lowest numbered test that evaluates to true, then the induction variable i is incremented by j and control passes to the node with the append statement. If none of the tests is true, control transfers to the first node. The second node performs an



(a) Dynamic schedule.

(b) Static SPL program.

Figure 7.5: Doacross applied to L.

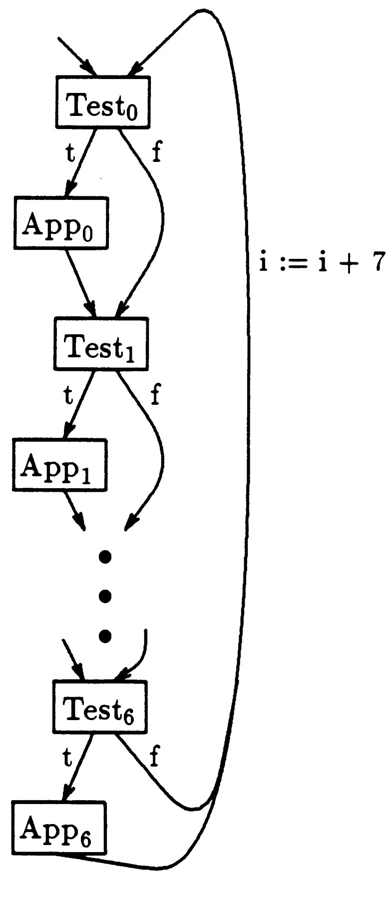


Figure 7.6: L unrolled seven times.

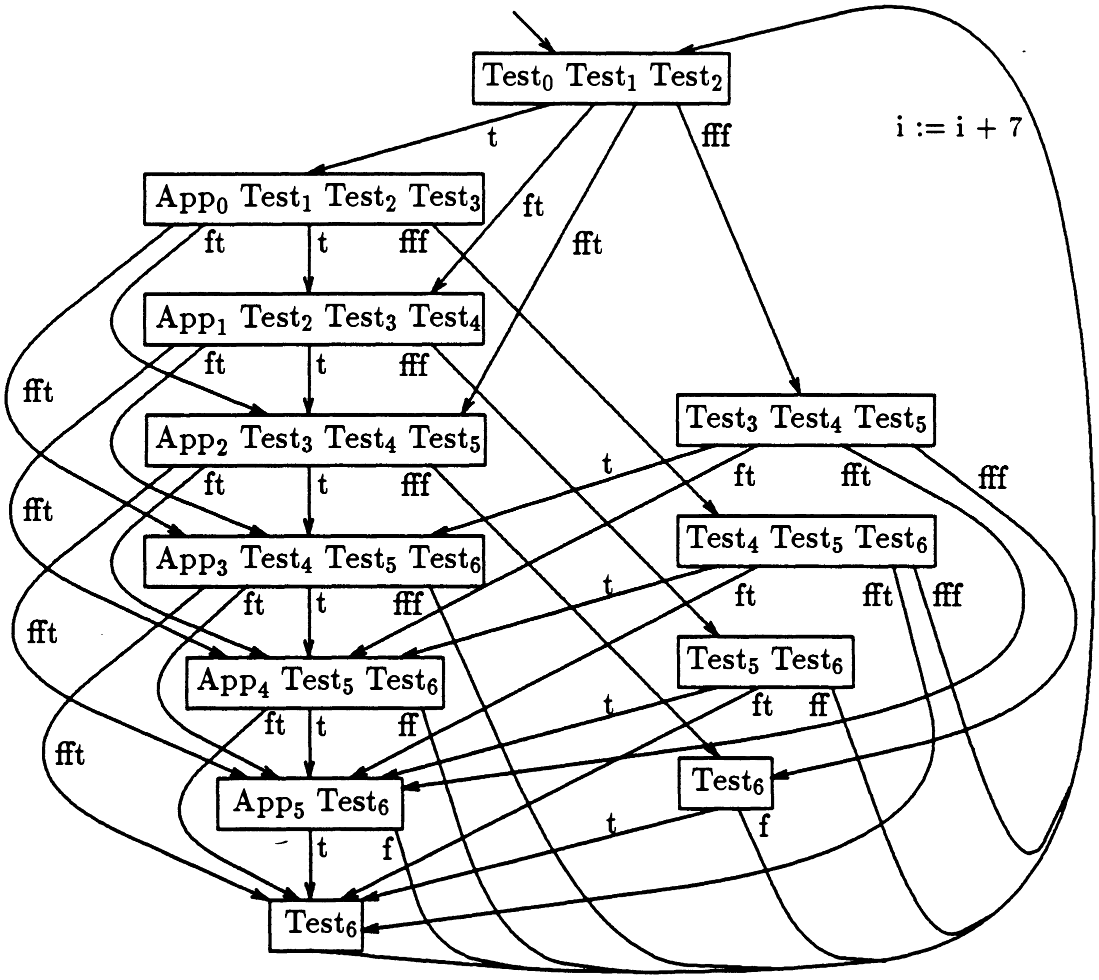


Figure 7.7: L unrolled seven times and compacted.

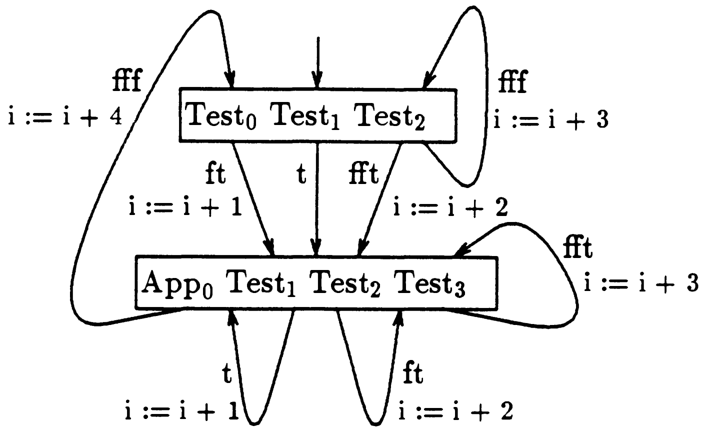


Figure 7.8: The same loop after pipelining.

append and evaluates the next three tests.

The pipelined loop executes three tests at every step, achieving optimal use of the critical resource. The final code can run on the TRACE, a commercial tightly-coupled parallel architecture that supports multiway jumps [Mul87]. In this example, the running time of perfect pipelining is dependent on the size (number of resources) of the machine as well as the original loop.

7.4 Comparison with Doacross

As suggested in the previous section, loops transformed by doacross can be represented in our formalism. In fact, a restriction on perfect pipelining with the simple rule corresponds exactly to doacross for single loops on synchronous multiprocessors. Another, more restrictive version corresponds to doacross for asynchronous multiprocessors. Thus, a family of transformations for different machine models can be directly formulated and compared in our framework.

The basic algorithm for doacross analyzes a loop body and decides where, on each path, it is safe to begin the next iteration. (We assume that statements in the loop body are not reorganized.) A communication instruction is added to the loop at those points. During execution, when a processor executing iteration i encounters a communication instruction, it sends a message signaling another processor that execution of iteration $i+1$ can begin.

Let $\mathcal{D}_{synch}$ be the compaction algorithm that implements doacross for synchronous multiprocessors. The restriction to the simple rule is made in move (see Figure 7.9). The new requirement is that if an iteration moves above a test, then it must move above that test on all paths. This restriction is necessary for doacross because the various processors have independent flow of control—once an iteration is started on a processor it must be able to proceed regardless of the path taken by any other processor. It is easily shown that for $\mathcal{D}_{synch}L$, the first statement of iteration $i+1$ overlaps iteration i exactly where the communications are introduced by doacross.

The asynchronous case ($\mathcal{D}_{asynch}$) is similar and can also be written as a restriction on the pipelining transformation; however, the communication points must be introduced even more conservatively because the processors do not run in lock-step. Again, the change is made in the move transformation. In $\mathcal{D}_{asynch}$, the first statement of iteration L_i may not move past a statement x in iteration L_{i-1} if any statement in L_i is dependent on x. This is necessary because processors may run at arbitrary speeds; thus, it is not safe to start an iteration L_i until everything L_i may depend on has already been executed. The following theorem summarizes the relationship between the three transformations.

Theorem 7.4.1 For all loops L, $\mathcal{D}_{asynch}L \leq_p \mathcal{D}_{synch}L \leq_p \mathcal{C}u^{\infty}L$.

Proof: Follows immediately from previous discussion and the fact that $\mathcal{D}_{asynch}$ is a restriction of $\mathcal{D}_{synch}$, which itself is a restriction of perfect pipelining with the simple rule. □

procedure *move*(x, n, m)
  if x is an assignment
    then
      begin
        if $x \in op(s)$ for all $s \in succ(n)$ **then**
          $P := move\text{-}op(P, x, succ(m), m)$
      end
    else $P := move\text{-}test(P, x, n, m)$;

Figure 7.9: The move transformation for $\mathcal{D}_{synch}$.

7.5 Efficiency

There are loops for which perfect pipelining with the simple rule requires time exponential in the size of the original loop. For example, consider a loop L with no loop-carried dependencies and a single test. If there are sufficient machine resources, then because there are no loop-carried dependencies the height of the program graph after compaction with the simple rule is a constant independent of the unrolling chosen. Now, u^iL has 2^i paths through the loop body; preserving all paths in constant depth requires exponential code explosion.

However, this can be detected after unrolling only once, because the iterations completely overlap after applying $\mathcal{C}$. In this case, the loop is completely vectorizable and generating good code is relatively easy.

Any compaction algorithm that is at least as powerful as the simple rule has worst-case exponential time complexity. In fact, every compaction-based parallelization system suffers from worst-case exponential performance [Ell85,Lam87, Ebc87,AN88a]. However, a good implementation can minimize the problem. Perfect pipelining often converges on many or all paths for unrollings much smaller than the worst case bound; thus interleaving unrolling, compaction, and the test for equivalent nodes substantially improves the efficiency of the algorithm. Another possibility is to dynamically modify the compaction algorithm if code explosion becomes too severe. By sacrificing some parallelism, code explosion can be substantially reduced while still guaranteeing the convergence of perfect pipelining.

Chapter 8

An Optimal Pipelining Algorithm

In this chapter, we develop a polynomial-time compaction algorithm for use with perfect pipelining. For loops with no tests in the loop body, the execution schedule obtained by this algorithm is *time optimal*: no transformation of the loop based on the given data-dependencies can yield a shorter running time for that loop.

The compaction algorithm we develop assumes unbounded resources; although for any loop the computed schedule has finite resource requirements, there are no *a priori* limits. We show that the optimal schedule can be used to derive a schedule that is within a factor of two of optimal for a machine with fixed resources.

8.1 The Problem

Much attention has been devoted to the parallelization of *doacross* loops [PH79, Cyt86,MS87]. A non-trivial doacross loop expresses some recurrence (a cycle of dependencies) preventing the iterations of the loop from executing independently. In this chapter we consider only non-trivial doacross loops; a variation on our technique generates optimal code for loops with data independent iterations in the presence of resource constraints [NPA88].

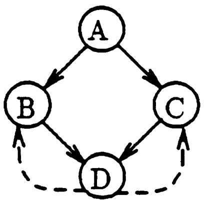
We assume that dependencies are given by a statically computed dependency graph (see Section 1.4). The nodes of the dependency graph are the statements of the loop, the edges connect nodes that may access the same memory locations [KKP+81]. In keeping with the standard dependency graph model in the literature, we assume that false dependencies must be observed. Thus, we need not deal with the incompleteness results of Chapter 4. We also assume that loop-carried dependencies have a threshold of one; an algorithm for bringing loops into this form is developed in Chapter 9.

These dependency assumptions are equivalent to the dependency analyzer D^1 of Section 6.6. Thus the algorithm computes a loop that is time optimal for D^1. As with the simple rule, if the loop-carried dependencies are caused by a limited form of indirect reference, we can show that the computed schedule is time-optimal with respect to D^∞. In particular, if all indirect references are array references of the form $A[I + c]$, where I is an induction variable and c is a constant, then an extension of this algorithm is time optimal with respect to D^∞.

The usual paradigm for parallelizing doacross loops is to execute the iterations of a loop on separate processors, subject to the constraint that loop-carried dependencies are not violated. A *delay* d is computed, such that if iteration i is begun d steps after iteration $i-1$, then loop-carried dependencies are preserved. A more general approach has also been studied: determining the optimal delay if dependency-preserving reorderings of the statements in the loop body are considered [MS87]. Computing a loop body with the optimal delay is NP-hard [Cyt84], so it would appear unlikely that a polynomial-time algorithm to compute optimal schedules exists. However, the intractability of the problem lies in the difficulty of finding the best ordering of statements in the loop body. This only shows

```
for i := 1 to N do
    begin
        A: A[i] := f1(B[i]);
        B: B[i] := f2(A[i], D[i - 1]);
        C: C[i] := f3(A[i], D[i - 1]);
        D: D[i] := f4(B[i], C[i])
    end;
```

(a) Original program.



(b) The dependency graph.

Figure 8.1: A sample loop.

that the general problem restricted to the case where iterations are scheduled as indivisible units is NP-hard.

A loop which illustrates this point is presented in Figure 8.1a; the dependency graph is given in Figure 8.1b. The loop-carried dependencies are shown as broken edges. A best doacross schedule is shown in Figure 8.2a; interchanging statements B and C is also a best doacross schedule. There is another schedule, shown in 8.2b, that is better.[1] This schedule issues a statement on the critical chain of dependencies at every step, and is therefore optimal with respect to the dependency graph. The algorithm we present computes this schedule.

Our major restriction is that the loop body can contain no tests except exit tests. There are techniques which essentially eliminate tests from a loop. *If-conversion* can be used to transform any loop into a semantically equivalent loop without tests [AKPW83]. Another method is to make assumptions about the

[1]Note that the schedule is better with respect to $\leq_p$—it is an improvement for a synchronous parallel machine. On an asynchronous multiprocessor it may or may not be better depending on communication costs.

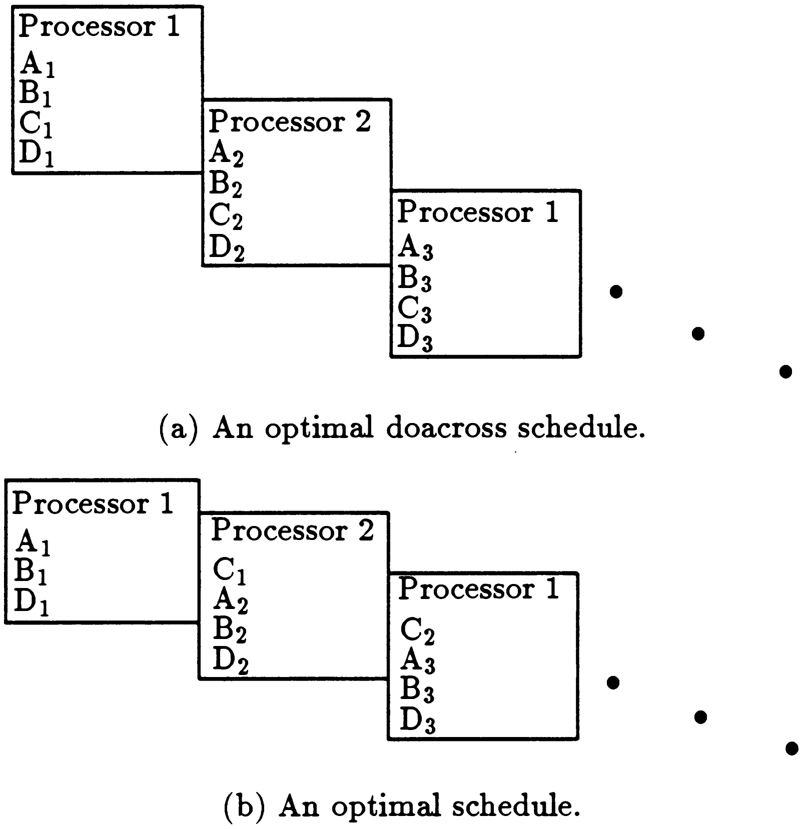


Figure 8.2: Possible schedules for the loop.

for each statement x **in** u^iP **do**
    migrate(x);

Figure 8.3: The greedy scheduling algorithm.

branching behavior of tests and apply a transformation to the most important path(s) [CP86,Fis81]. It is an open problem whether there is an algorithm to compute optimal parallel loop schedules without making such strong assumptions about flow of control and using only transformations that preserve data dependencies.

8.2 The Approach

The basic strategy behind our method is very simple. The loop is repeatedly unrolled and compacted as much as possible. An algorithm for doing this, called *greedy scheduling*, is given in Figure 8.3. If the longest chain of dependencies on which a statement x depends has length j, then x is scheduled in the jth node of the compacted loop. A loop's dependencies have some regularity, specified by the dependency graph. The results of Chapter 6 show that scheduling a large enough portion of a loop's execution history reveals some repeating behavior, which (intuitively) can be used to obtain a good schedule for the loop. As in Chapter 6, we refer to this repeating behavior as the *pattern* of the loop. We show that the portion of the execution history that must be examined to compute optimal schedules is small. This yields a polynomial-time algorithm for computing optimal parallel schedules for a large class of loops and parallel machines.

The difficulty with this approach is that greedy scheduling is not a bounded compaction algorithm; thus, perfect pipelining does not converge when used with greedy scheduling. Greedy scheduling does, however, provide important information about an optimal parallel schedule for the loop. We show that after scheduling a small number of iterations the critical dependency chains of the loop can be detected. Furthermore, greedy scheduling is time optimal for the (finite) number of iterations it schedules. The information about critical chains is used to dynamically modify greedy scheduling so that it becomes a bounded compaction algorithm while preserving optimality.

8.3 Computing Patterns for Statements

We first show that the occurrences of an individual statement exhibit a pattern once a sufficient number of iterations are scheduled. In what follows, L is an innermost loop containing n statements. A reference to the dependency graph indicates the dependency graph of L. The occurrence of statement x in iteration i is denoted x_i; superscripts distinguish distinct statements.

Definition 8.3.1 A *dependency chain* is a sequence of statements $x^1_{h_1} x^2_{h_2} \dots x^k_{h_k}$ such that (x^i, x^{i+1}) is an edge in the dependency graph.

Definition 8.3.2 Let $C = x^1_{h_1} \dots x^k_{h_k}$ be a dependency chain.

- C is a *cycle* if $x^1 = x^k$.
- The *length* of C, written $|C|$, is k.
- C' is a *subchain* of C if $C' = x^i \dots x^j$ where $1 \le i \le j \le k$.
- The *span* of C is the number of iterations $h_k - h_1$.
- C *reaches* statement y if $x^i_{h_i}$ is scheduled in node i, y is scheduled in node $k+1$, and (x^k, y) is an edge in the dependency graph.

Definition 8.3.3 (Cost) Let $C = x^1 \dots x^k x^1$ be a cycle. Let p be the number of loop-carried dependencies—including the dependency (x^k, x^1)—in the cycle. The *cost* of C is the ratio k/p.

The cost of a cycle establishes a lower bound on the rate that statements in the cycle can be executed. Using C from Definition 8.3.3, statement x^i_{j+p}

cannot execute earlier than k steps after x_j^i. (This notion of the cost of a cycle is due to Callahan, Cocke, and Kennedy [CCK87].) In a greedy schedule, the two statements must be scheduled at least k nodes (steps) apart. We write $dist(x, y)$ for the number of nodes separating x and y in a greedy schedule. Let $cost(x) = k_x/p_x$ be the maximum cost of any cycle on which statement x depends. If x is not dependent on any cycle, then $cost(x) = 0/1$. We show that, in a greedy schedule, after scheduling $O(n^2)$ iterations, any subsequent occurrences of a statement x are scheduled exactly k_x nodes after the occurrence of x p_x iterations before. Thus, a pattern for each statement can be inferred from a greedy schedule of at most $O(n^2)$ iterations. The following lemmas are required to prove this result.

Lemma 8.3.4 Let C be a chain with $|C| \geq i(n+1)$ for some positive i. Then there are at least i disjoint subchains of C that are cycles.

Proof: Partition C into subchains $C_1, C_2, \ldots, C_i$, where $|C_j| > n$. Each C_j must contain a cycle, as there are only n distinct statements. □

Lemma 8.3.5 Given p integers $a_1, \ldots, a_p$, there is a subset S of the a_i such that

$$\sum_{a_i \in S} a_i \equiv 0 \bmod p$$

Proof: Let $s_i = (a_1 + \ldots + a_i) \bmod p$. If all of the s_i are distinct, then one must be zero, as there are only p distinct numbers. If s_i and s_{i+j} are equal, then $0 \bmod p \equiv s_{i+j} - s_i = a_{i+1} + \ldots + a_{i+j}$. □

Theorem 8.3.6 Let x be a statement with cost k/p, and let the loop body contain n statements. In a greedy schedule, in any iteration i greater than $2np + 3p$, $dist(x_{i-p}, x_i) = k$.

Proof: We first prove the theorem for statements x which are members of the cycle of maximum cost on which they depend. Assume for some $i > np + 2p$ that $dist(x_{i-p}, x_i) > k$. Let C be a chain reaching x_i. There are two cases:

- $Span(C) \leq np + p$ iterations. Let C' be a chain reaching x_{i-p}. From the dependency graph, a chain of dependencies identical to C reaches x_{i-p}. But $|C'| + k \leq |C|$, a contradiction.

- $Span(C) > np + p$ iterations. By Lemma 8.3.4, there are at least p disjoint subchains of C that are cycles. By Lemma 8.3.5, there is a subset of these cycles $\{C_h\}$ such that $\sum_h |C_h| = jp$ for some $j > 0$. Deleting the cycles $\{C_h\}$ from C produces a chain C' which reaches x_y, where $y = i - jp$. By assumption, there is a chain of length jk from x_y to x_i. But $\sum_h |C_h| \leq jk$, or else some C_h has cost greater than k/p, a contradiction. Therefore $|C'| + jk \geq |C|$, implying that $dist(x_y, x_i) = jk$. Since $dist(x_y, x_{i-p}) \geq (j-1)k$, $dist(x_{i-p}, x_i) \leq k$.

Now consider a statement y which is not a member of the cycle C of maximum cost on which it depends. Let $x_i D y_k$ be a chain reaching y_k, where x is a member of C and D contains no occurrence of x. By the regularity of dependencies, we need only determine the length of the shortest possible D.

We claim that there is a D such that $|D| \leq np + p$. Assume not. By Lemmas 8.3.4 and 8.3.5, we can delete subcycles from D of total length jp for some positive j. The resulting chain D' satisfies $x_i D' y_{k-jp}$. By the regularity of dependencies, this implies that there is a chain $x_i C \ldots C x_{i+jp} D' y_k$. Now, $|x_i C \ldots C x_{i+jp} D' y_k| \geq |x_i D y_k|$, or else one of the cycles deleted from D has a cost greater than C, a contradiction. Thus, the chain $x_i C \ldots C x_{i+jp} D'$ reaches y_k. If $|D'| > np + p$, cycles can be deleted from D'; iterating this procedure produces a chain of length no more than $np+p$ that satisfies the assertion. After $(np + 2p) + (np + p) = 2np + 3p$ iterations are scheduled, $dist(x_{i-p}, x_i) = k$. □

Theorem 8.3.6 shows that after scheduling $O(np_x)$ iterations, occurrences of x are scheduled at a constant (and optimal) rate. Thus, a sufficiently large greedy schedule exhibits a pattern for each statement.

Corollary 8.3.7 After scheduling $O(n^2)$ iterations every statement is scheduled at the optimal rate. Furthermore, if $p_x \leq 1$ for all x, then $O(n)$ iterations are sufficient.

Corollary 8.3.7 follows from Theorem 8.3.6 since $p_x \leq n$ for any x. The special case where $p \leq 1$ is important in practice: we have examined the inner loops of the Eispack routines [SBI+70] and the Livermore Loops [McM83] and have not found a single example where $p > 1$.

The efficiency of our algorithm is dependent on the cost of computing a greedy schedule. Thus far, we have cast the algorithm in terms of the core transformations. However, because of the special structure of the problem there are more direct and efficient methods. Using a modified topological sort of the dependency graph, the cost of greedy scheduling is proportional to the number of statements scheduled; thus, to schedule $O(n^2)$ iterations of n statements requires $O(n^3)$ time; if $p \leq 1$ then only $O(n)$ iterations must be scheduled, requiring $O(n^2)$ time.

8.4 A Bounded Algorithm

There are two drawbacks to the simple greedy scheduling algorithm. First, to be assured that the pattern for each statement has been detected, it is necessary to run for $O(n^3)$ time. As observed above, in practice the pattern for each statement emerges much earlier, in $O(n^2)$ time or less. Second, greedy scheduling is not a bounded compaction algorithm.

We use greedy scheduling to derive another compaction algorithm that is bounded and converges as quickly as possible. The idea is to reschedule statements not on the critical path so that they have the same cost as statements on the critical path. This yields a very compact pattern for the entire loop.

We illustrate Theorem 8.3.6 with a sample loop due to Cytron [Cyt84]. The loop is shown in Figure 8.4; the dependency graph is given in Figure 8.5a. Figure 8.5b shows the greedy schedule of five iterations. For clarity, we drop the

```
for i := 1 to N do
    begin
        A: A1[i] := B[i];
        B: A2[i] :=                  A8[i - 1];
        C: A3[i] :=                  A5[i - 1];
        D: A4[i] := A3[i]            +A7[i - 1];
        E: A5[i] := A2[i];
        F: A6[i] := A1[i]            +A13[i - 1];
        G: A7[i] := A4[i];
        H: A8[i] := A4[i] + A5[i]    +A17[i - 1];
        I: A9[i] := A1[i];
        J: A10[i] := A9[i]           +A15[i - 1];
        K: A11[i] := A9[i];
        L: A12[i] := A9[i];
        M: A13[i] := A12[i];
        N: A14[i] := A13[i];
        P: A15[i] := A14[i];
        Q: A16[i] := A14[i];
        R: A17[i] := A14[i]
    end;
```

Figure 8.4: A sample loop.

flow graph representation of a program. Iterations have been listed separately, side by side. The vertical axis is time; all statements on a horizontal line of the figure are members of the same program node. For simplicity, the loop control code has been omitted.

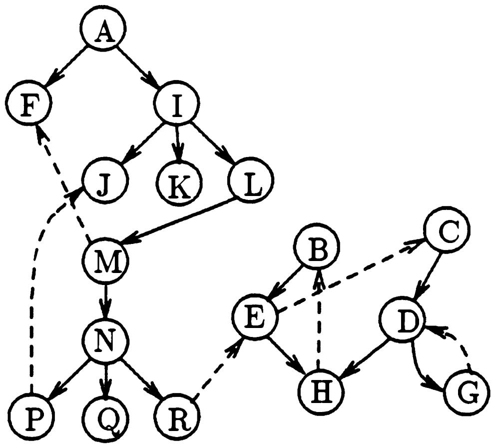
In this example, the cycle B, E, H has the greatest cost (three). Statements C, D, and G are dependent on this cycle and thus have the same cost. All other statements have cost 0/1. In Figure 8.5b, the code is split into two groups that repeat every iteration, one with a cost of three, the other with a cost of zero.

In Figure 8.5b, note that the statements with cost 0/1 in iterations three and four can be delayed without affecting the length of the schedule. Eliminating the "gaps" in the iterations—intervals of nodes with no statements from that iteration—produces the schedule in Figure 8.6. The boxed area is the pattern for the loop; scheduling additional iterations (without gaps) reproduces these three nodes. Note that no statement on the critical path has been delayed as a result of rescheduling statements. We derive a method for determining when statements can be delayed without affecting the optimality of the final schedule.

When an iteration is scheduled, it is spread across some interval of nodes $n_1, \ldots, n_k$. As in Figure 8.5b, statements from an iteration tend to cluster into groups of mutually dependent statements with gaps between the groups.

Definition 8.4.1 (Region) A *region* of a scheduled iteration is an interval of nodes $A = n_1, \ldots, n_k$ such that some statement from the iteration is scheduled in every n_j.

We are interested in the maximal regions of an iteration. Between any two adjacent maximal regions there must be at least one node with no statement from that iteration. These gaps play a critical role in detecting a pattern. In Figure 8.5b, there are two maximal regions for the last iteration: nodes $1, \ldots, 7$ and $13, \ldots, 16$.



(a) The dependency graph.

	iteration				
time	0	1	2	3	4
1	ABC	A	A	A	A
2	DEFI	I	I	I	I
3	GHJKL	CKL	KL	KL	KL
4	M	BDM	M	M	M
5	N	EFGN	FN	FN	FN
6	PQR	PQR	CPQR	PQR	PQR
7		HJ	DJ	J	J
8			BG	—	—
9			E	—	—
10			H	C	—
11				BD	—
12				EG	—
13				H	C
14					BD
15					EG
16					H

(b) The code after five phases.

Figure 8.5: Greedy scheduling.

Definition 8.4.2 (Gaps) Let $A_1, \ldots, A_j$ be the maximal regions of an iteration, where $A_i = n_i, \ldots, n_{i'}$ and $n'_i < n_{i+1}$ for all i. Then $gap(A_i, A_{i+1}) = n_{i+1} - n'_i$.

In Figure 8.5b, iterations three and four have the same maximal regions; the only difference between the iterations is the size of the gap. In this sense, iteration four is a "stretched-out" version of iteration three. We say that two scheduled iterations i and $i+c$ are *alike* if they have the same maximal regions and the inter-region gaps in iteration $i+c$ are as large or larger than in iteration i. We present conditions under which the gaps in the larger iteration can be shrunk to the size of the gaps in the smaller iteration without affecting the optimality of the final code.

Theorem 8.4.3 Consider $i+c$ scheduled iterations, where iteration i and $i+c$ are alike, with j maximal regions. Assume there is no unbroken chain of dependent statements from a statement in A_k of iteration i ($k < j$) to a statement in A_j of iteration $i+c$. Then the inter-region gaps in iteration $i+c$ can be reduced to the size of the corresponding gaps in iteration i and for any additional (greedily) scheduled iterations the resulting schedule is optimal.

Proof: After shrinking the gaps iteration $i+c$ is identical to iteration i. Then (greedily) scheduled iterations $i+c$ to $i+2c$ in the new schedule are identical to iterations i to $i+c$ in the original schedule. We claim there is no shorter schedule for $i+2c$ iterations. For iterations i to $i+c$, the critical chain of dependencies is between the regions A_j of iteration i and A_j of iteration $i+c$. By symmetry, the critical chain of iterations $i+c$ to $i+2c$ is between the regions A_j of iteration $i+c$ and A_j of iteration $i+2c$. This implies no statement in A_j of iteration $i+2c$ is delayed by shrinking the gaps in iteration $i+c$. Applying this argument inductively shows that any larger greedy schedule is also time optimal. □

In Figure 8.5b, iterations three and four are alike and no chain from the first region of iteration three reaches to the second region of iteration four. Thus it is

	iteration						
time	0	1	2	3	4	5	6
1	ABC	A	A	—	—	—	—
2	DEFI	I	I	—	—	—	—
3	GHJKL	CKL	KL	A	—	—	—
4	M	BDM	M	I	—	—	—
5	N	EFGN	FN	KL	—	—	—
6	PQR	PQR	CPQR	M	A	—	—
7		HJ	DJ	FN	I	—	—
8			BG	PQR	KL	—	—
9			E	J	M	A	—
10			H	C	FN	I	—
11				BD	PQR	KL	—
12				EG	J	M	A
13				H	C	FN	I
14					BD	PQR	KL
15					EG	J	M
16					H	C	FN
17						BD	PQR
18						EG	J
19						H	C
20							BD
21							EG
22							H

Figure 8.6: The pattern.

safe to shrink the gap in iteration four to the size of the gap in iteration three. Below, we describe a simple test to determine when it is safe to completely close the gaps between regions; in this case, the gaps can be eliminated. Figure 8.6 shows the schedule of Figure 8.5b extended with a few additional iterations and with the gaps completely closed.

8.4.1 Gaps are Required

The results of Chapter 6 show that the rate of convergence of a pipelining algorithm is dependent on the number of classes of equivalent nodes that the algorithm produces. The larger the gaps introduced by the modified greedy scheduling algorithm, the more classes of equivalent nodes there are, and the slower the rate of convergence. Therefore, it is important to know whether gaps must be introduced in optimal schedules. We show that, in general, any algorithm that computes optimal schedules must introduce gaps in scheduled iterations.

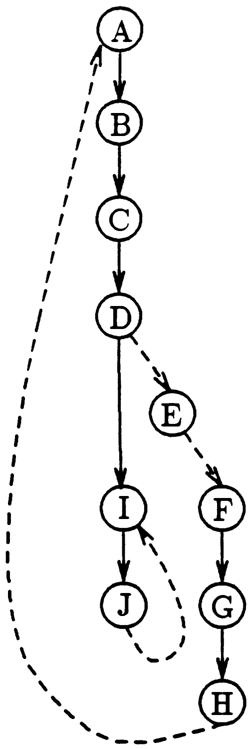
An example is given in Figure 8.7. In the figure, the cycle A, ..., H has a cost of 8/3 and the cycle I, J has a cost of 2, so A, ..., H is the critical cycle. Eliminating the gap in iteration two of the greedy schedule must delay statement D, E, or H, each of which is on a critical path. Thus, an algorithm which produces optimal schedules must, in general, introduce gaps in the execution of iterations. The following theorem shows when completely closing gaps preserves optimality.

Theorem 8.4.4 Let iterations i and $i+c$ be alike. Assume that all iterations i through $i+c$ have the same number of maximal regions. Let A_j^k be the jth maximal region of iteration k, where $i \le k \le i+c$. If

$$\forall j, k \; gap(A_j^k, A_{j+1}^k) \le gap(A_j^{k+1}, A_{j+1}^{k+1})$$

then the gaps in iteration $i+c$ (and i) may be completely closed.

Proof: Because the gaps are always increasing between consecutive iterations, completely closing the gaps in iteration i and rescheduling the iterations $i+1$



(a) A dependency graph.

	iteration		
time	0	1	2
1	AEF	—	—
2	BG	F	—
3	CH	G	—
4	D	AH	—
5	I	BE	A
6	J	C	BF
7		D	CG
8		I	DEH
9		J	—
10			I
11			J

(b) A greedy schedule of three iterations.

Figure 8.7: A situation where gaps are required.

```
(* L is the original loop. *)
repeat
    unroll one iteration L_i of L;
    for each statement x in L_i do
        migrate(x);
    for k := 0 to i - 1 do
        if L_k and L_i satisfy Theorem 8.4.3 then
            shrink gaps in L_i;
until convergence;
```

Figure 8.8: The modified greedy scheduling algorithm.

through $i+c$ accordingly does not delay any statement on a critical path. Therefore, because i and $i+c$ are alike, completely closing the gaps in $i+c$ does not delay any statement on a critical path in any larger greedy schedule. □

8.4.2 The Algorithm and Its Correctness

The modified greedy scheduling algorithm ($\mathcal{O}$) is outlined in Figure 8.8. We show that perfect pipelining combined this algorithm produces time optimal schedules.

Theorem 8.4.5 Let L be a loop without tests in the loop body, and let the dependencies of L be given by a dependency graph. Then $\mathcal{O}u^{\infty}L$ is time optimal.

Proof: The modified greedy scheduling algorithm is bounded and local, so $\mathcal{O}u^{\infty}L$ exists. The algorithm never delays any statement in a critical chain of dependencies, so $\mathcal{O}u^{i}L$ is time optimal for i iterations. But $\forall i\ \mathcal{O}u^{i}L \leq_p \mathcal{O}u^{\infty}L$; thus, by Lemma 2.3.10, $\mathcal{O}u^{\infty}L$ is time optimal for any number of iterations. □

The conditions of Theorem 8.4.3 require a check that iterations i and $i+c$ are alike, and that no chain of dependencies from a region in iteration i (except the last region) reaches the last region of iteration $i+c$. For efficiency, c should be minimized, because implementing the dependency chain check is expensive relative to the cost of greedy scheduling. If $p_x \leq 1$ for all statements in the loop, then checking consecutive iterations ($c = 1$) is sufficient. Consecutive iterations satisfy Theorem 8.4.3 if they are alike—no explicit dependency chain analysis is required. By Theorem 8.4.4, if consecutive iterations are alike the gaps can be completely closed. (Put another way, if $p \leq 1$ for all statements, then there is an optimal schedule without gaps in iterations.) Because $p \leq 1$ in practice, this yields an algorithm that runs in $O(n^2)$ in most cases.

In principle, there are loops for which this strategy of making iterations "look alike" cannot succeed in polynomial time. Let the denominator of $cost(x)$ be the *period* of x. The length of a pattern computed by the modified greedy scheduling algorithm is at least the least common multiple of the periods of the statements in the loop. If many statements have large and relatively prime periods, this is potentially exponential in n. We believe that such loops are extremely unlikely to be written. As mentioned above, in practice p_x for a statement x is zero or one. Even if there are "real" loops where $p_x > 1$, one would not expect such loops to have many arbitrarily large and relatively prime statement periods. Recall that after $O(n^3)$ time the costs (and therefore periods) of all statements are known; thus loops for which the pattern involves more than adjacent iterations can be detected and handled at that point and the size of the pattern resulting from delaying statements can be computed exactly.

8.5 Mapping Optimal Schedules to Processors

The transformed loop consists of a prologue (everything before the pattern), the pattern (with a backedge from the last step in the pattern to the first step in the pattern), and an epilogue (everything after the pattern). The final loop for the loop in Figure 8.4 is shown in Figure 8.9.[2] The subscripts indicate the increment to the induction variable i.

If the number of processors available for execution of the loop is at least the

[2]Including the epilogue requires some adjustment of the loop bounds. The details are presented in Chapter 11.

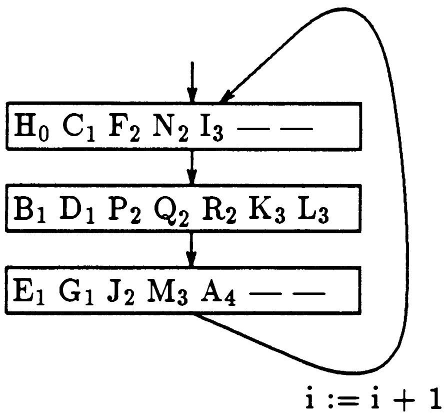


Figure 8.9: An optimal schedule.

maximum width of the pattern discovered by our algorithm, then the loop can be run in this raw form. Simple heuristics can reduce the width of the pattern without increasing its length. For example, in Figure 8.5a statement L has no dependents, and so can be delayed until the third node of the loop in Figure 8.9a, reducing the width from seven to six.

If the target machine M has fewer than the number of processors required by the optimal schedule, then the schedule is not legal for that machine. We show that even a naive heuristic algorithm can use the optimal schedule to generate code that is within a factor of two of optimal for M.

Let M have t processors, and let L be a time-optimal loop that requires $t' > t$ processors. Let k be the number of nodes in L, and let t_i be the number of statements in node n_i of L. A node is *full* if it has t statements. For each node n_i, create $\lceil t_i/t \rceil$ nodes, where $\lfloor t_i/t \rfloor$ nodes are full and the last is (potentially) partially full. Replace n_i with these $\lceil t_i/t \rceil$ nodes. Let the resulting loop be L'.

Lemma 8.5.1 At most k nodes in L' are not full.

Proof: If a node in L is not full, then it contributes one partially full node to L'. Similarly, if a node in L has more than t statements, it contributes at most one partially full node to L'. So the total number of nodes in L' that are not full is at most k. $\square$

Let $T_{L'}$ be the running time of loop L', and let T_L be the running time of loop L. Let T_* be the optimal running time of loop L rescheduled to run on a machine with t processors.

Theorem 8.5.2 $T_{L'}/T_* \leq 2$

Proof: Let k' be the number of nodes in L'. Clearly $T_L \leq T_*$. If $k' \leq 2k$ then $T_{L'}/T_L \leq 2$ and the theorem holds. Assume $k' > 2k$. By Lemma 8.5.1, at least half the nodes of L' are full, implying $T_{L'}/T_* \leq 2$. $\square$

8.6 Experiments

The implementation of the optimal scheduling algorithm takes into account instruction latencies and processor limitations. The optimality results hold when instruction latencies are greater than one. To measure the performance of our technique, we have tested it on the first fourteen Livermore Loops [McM83] assuming hardware similar to the Cray 1. The results are encouraging: the execution rate for the entire benchmark, as measured by the harmonic mean, ranges from 20 MFlops (millions of floating point operations per second) for a pipelined uniprocessor to 53 MFlops for a VLIW with unlimited resources. Average performance (which tends to weight the benchmark programs that parallelize easily) ranges from 28 MFlops for a pipelined uniprocessor to 537 MFlops for a VLIW with unlimited resources. In contrast, the unparallelized code achieves 7 MFlops by the harmonic mean, 8 MFlops by the average. A complete report of the results and the experiment are in [AN88b].

Chapter 9

Loop Quantization

Loop unrolling is an important component of our transformational system. Loop unrolling helps exploit fine-grain parallelism by providing a large number of statements (the unrolled loop body) for scheduling. In this chapter we extend loop unrolling to multiple nested loops using *loop quantization* [Nic85a,AN87a].

This chapter is divided into four major sections. Section one describes and presents correctness conditions for loop quantization. Section two presents an abstract representation of loops that captures the information required for quantization. In section three this representation is used to prove a number of properties of loop quantization and to develop an algorithm. Section four develops a generalization of loop quantization, *mitred quantization.* The appendix contains details of the algorithms and dependency analysis.

9.1 Unrolling Multiple Loops

The most direct method for unrolling nested loops is to fully unroll each loop from innermost to outermost. When a loop is fully unrolled, no artificial constraints are introduced by the indexing order; limitations on parallelization are imposed only by data dependencies. Full unrolling is normally infeasible because loop bounds are usually unknown at compile time. However, even if the amount of

```
for i := 0 to n do
    for j := 0 to n do
        begin
            x := A[i + 16, j];
            y := A[i + 1, j];
            z := x + y;
            A[i + 1, j + 1] := z
        end;
```

Figure 9.1: A loop L.

unrolling is limited, the ability to unroll all nested loops is necessary. Unrolling the innermost loop exposes parallelism only in that loop. This is not satisfactory, because parallelism is often available between several of the outer loops.

In scientific programs, the central difficulty in parallelizing nested loops is dealing with indirect references. We focus on the case of multiple nested loops with bounded iteration (e.g., DO loops in Fortran) containing array references. We use the nested loops in Figure 9.1 as an example. In general we consider n nested loops, where loops are labeled L_1 (outermost) to L_n (innermost). An *iteration variable* I_l and an *increment* K_l are associated with each loop L_l. Values of I_l and K_l are denoted by i_l and k_l respectively. Initially, loops are assumed to be normalized to range from 0 to N_l with increments of 1; an algorithm for bringing loops into this form is well-known [Wol78].

In the following discussion, we assume that the n loops are the only ones affected by quantizations. An *iteration vector* [Kuh80] is an assignment of the iteration variables $\langle i_1, \ldots, i_n \rangle$. Iteration vectors are ordered lexicographically. The *iteration space* is the set of all iteration vectors of a loop.

Multiple loops are unrolled, in order, from L_n to L_1. To unroll loop i k_i times, the loop body is duplicated k_i times. This is equivalent to unrolling all nested loops fully when the upper bounds are $k_1, \ldots, k_n$. In what follows, we write L for the original loops and $\hat{L}$ for the quantized loops.

```
for i := 0 to n do
    for j := 0 to n by do
        begin
            x := A[i + 16, j];
            y := A[i + 1, j];
            z := x + y;
            A[i + 1, j + 1] := z;
            x := A[i + 16, j + 1];
            y := A[i + 1, j + 1];
            z := x + y;
            A[i + 1, j + 2] := z;
            x := A[i + 16, j + 2];
            y := A[i + 1, j + 2];
            z := x + y;
            A[i + 1, j + 3] := z;
            ⋮
            x := A[i + 16, j + 9];
            y := A[i + 1, j + 9];
            z := x + y;
            A[i + 1, j + 10] := z
        end;
```

Figure 9.2: The inner loop unrolled ten times.

```
for i := 0 to n by 10 do
    for j := 0 to n by 10 do
        begin
            x := A[i + 16, j];
            y := A[i + 1, j];
            z := x + y;
            A[i + 1, j + 1] := z;
            ⋮
            x := A[i + 16, j + 10];
            y := A[i + 1, j + 10];
            z := x + y;
            A[i + 1, j + 11] := z;
            x := A[i + 17, j];
            y := A[i + 2, j];
            z := x + y;
            A[i + 2, j + 1] := z;
            ⋮
            x := A[i + 17, j + 9];
            y := A[i + 2, j + 9];
            z := x + y;
            A[i + 2, j + 10] := z;
            ⋮
            x := A[i + 25, j];
            y := A[i + 10, j];
            z := x + y;
            A[i + 10, j + 1] := z;
            ⋮
            x := A[i + 25, j + 9];
            y := A[i + 10, j + 9];
            z := x + y;
            A[i + 10, j + 10] := z
        end;
```

Figure 9.3: The quantized loop $\hat{L}$.

For example, in Figure 9.2 the inner loop of Figure 9.1 is unrolled ten times. To make the examples clear (and realistic), intermediate exit tests and assignments to the iteration variables are eliminated. The details of how this is done are given in Section 9.5. Figure 9.3 shows the original loop after the outer loop is also unrolled ten times. The iteration space of $\hat{L}$ consists of 10×10 "boxes" of iterations of L. All unrolled iterations in the box are executed before the box is shifted (by a quantum jump) along any dimension.

When outer loops are unrolled, $\hat{L}$ may not preserve the semantics of L because the execution order of the iterations is altered. For example, in Figure 9.3, original loop iteration $\langle 9,9 \rangle$ is executed before original loop iteration $\langle 0,10 \rangle$ in $\hat{L}$. We present conditions under which $\hat{L}$ preserves the semantics of L.

Definition 9.1.1 Let $\hat{L}$ be a quantization of L. Let w_x be an iteration vector when a memory location x is written in L, and let r_x be an iteration vector when x is read in L. Similarly, let $\hat{w}_x$ be the corresponding iteration vector when x is written in $\hat{L}$, and let $\hat{r}_x$ be the corresponding iteration vector when the x is read in $\hat{L}$. Then $\hat{L}$ is a legal quantization if

$$\begin{aligned} \forall x, w_x, r_x \qquad & r_x < w_x \Rightarrow \hat{r}_x \leq \hat{w}_x \wedge \\ & r_x > w_x \Rightarrow \hat{r}_x \geq \hat{w}_x \end{aligned}$$

If these conditions are satisfied, then $\hat{L}$ preserves the order of reads and writes in L.[1] Note it is possible that $\hat{r}_x = \hat{w}_x$ even though $r_x \neq w_x$. This is because each iteration of $\hat{L}$ contains many iterations of L. The unrolling preserves the execution order of dependent statements within an iteration of $\hat{L}$.

Lemma 9.1.2 If $\hat{L}$ is a legal quantization of L, then L **sim** $\hat{L}$.

Proof: Immediate from Definition 9.1.1. □

[1] In general, dependencies between writes (output dependencies [KKLW80] or, more generally, def-order dependencies [HPR88b]) must also be considered. Our formalism applies without modification to these dependencies; they are omitted to simplify the presentation.

If a quantization is illegal because of conflicting references, it may be possible to increase the size of the quantization to include any dependent references. This may require unrolling a loop completely. In some applications one loop—of several nested loops—has a known, small number of iterations; hence, full unrolling on that loop may be feasible. In cases where full unrolling is impractical, limited unrollings must be chosen to ensure correctness. In Figure 9.1, the outer loop can be unrolled fifteen times, but not more. To see this, let x be the location $A[16,1]$. In L, x is written in iteration $\langle 15,0\rangle$ and read in iteration $\langle 0,1\rangle$. If the outer loop is unrolled sixteen times, then in $\hat{L}$ x is written in iteration $\langle 0,0\rangle$ and read in iteration $\langle 0,1\rangle$; the order of the references is reversed and the quantization is illegal.

9.2 An Abstraction of Nested Loops

Constraint 9.1.1 describes legal quantizations, but does not suggest any algorithm for computing quantizations. In this section we develop an abstract representation that captures information about loop-carried dependencies [All83]. Our representation is general and is particularly useful in loop quantization algorithms. We also prove a number of properties of loop quantization using the abstraction.

The abstraction consists of a *comparison* table and a *distance* table for L. The comparison table captures the ordering of read and write references across iterations. The distance table describes the "distance" (in numbers of iterations) between conflicting references.

In what follows, L is a set of n nested loops, A is an n-dimensional array, and $A[f_1,\ldots,f_n]$ is a reference to A where f_j is an expression for the jth subscript of A using iteration variable i_j. The appendix includes details of how the tables can be computed for commonly occurring references. We assume for simplicity that all arrays have the same number of dimensions.

Comparison tables are based on the *direction vectors* of Wolfe [Wol78]. Comparison tables have also been defined independently by Kennedy [Ken87]; he uses *direction matrices* to explore vectorization. Distance tables are based on the *distance vectors* (also called *difference vectors*) of Lamport [Lam74]. Cytron makes use of what are essentially distance tables to develop an algorithm for computing delays in doacross loops [Cyt86].

9.2.1 Comparison Tables

Definition 9.2.1 The *time* at which an element $\langle j_1, \ldots, j_n \rangle$ of array A is addressed by the reference $A[f_1, \ldots, f_n]$ is the vector $\langle f_1^{-1}(j_1), \ldots f_n^{-1}(j_n) \rangle$, if all of the inverses exist. Times are ordered lexicographically.

Loop quantization requires that the read and write references of an array in L be compared; we present a concise notation for comparing references. We assume that the upper and lower bounds of iteration variables are unknown, and that array references contain no constant references (e.g., $A[5]$). Thus, the inverses in Definition 9.2.1 are always well-defined. These restrictions are included to make the development concise; the technique can be extended to accommodate constant references and knowledge of upper and lower loop bounds.

A quantization is correct if it preserves the order of reads and writes for each memory location. We begin, therefore, by defining a notation for discussing the order of reads and writes for a particular location.

Definition 9.2.2 Let $A[r]$ and $A[w]$ be a read and write reference (respectively) of a one-dimensional array A in L. Let p be the pair $\langle r, w \rangle$. Then x^p, the *order* of reads and writes at location x with respect to p, is:

$$x^p = \begin{cases} =_o & \Leftrightarrow \quad r^{-1}(x) = w^{-1}(x) \wedge I \\ \neq_o & \Leftrightarrow \quad r^{-1}(x) \notin \mathcal{N} \vee w^{-1}(x) \notin \mathcal{N} \\ >_o & \Leftrightarrow \quad r^{-1}(x) > w^{-1}(x) \wedge I \\ <_o & \Leftrightarrow \quad r^{-1}(x) < w^{-1}(x) \wedge I \end{cases}$$

where I is the predicate $r^{-1}(x) \in \mathcal{N} \wedge w^{-1}(x) \in \mathcal{N}$

If x^p is $=_\mathrm{o}$ then x is both read and written in the same iteration. The relation $<_\mathrm{o}$ specifies x is read before it is written, $>_\mathrm{o}$ that it is written before it is read. The relation $\neq_\mathrm{o}$ indicates that there is no conflict at x. The generalization to multi-dimensional arrays is straightforward.

Definition 9.2.3 Let $A[r_1, \ldots, r_n]$ and $A[w_1, \ldots, w_n]$ be a read and write reference of A in L. Let p be the vector $\langle p_1, \ldots, p_n \rangle$ where $p_k = \langle r_k, w_k \rangle$. If $x = \langle x_1, \ldots, x_n \rangle$ is a location in array A, then $x^p = \langle x_1^{p_1}, \ldots, x_n^{p_n} \rangle$.

Having categorized the possible read/write orderings for a pair of references with respect to a single location, we can express the possible read/write orderings for a pair of references over all possible locations.

Definition 9.2.4 Let $B[r]$ and $B[w]$ be a read and a write reference (respectively) of a one-dimensional array B in L. $B[r]$ and $B[w]$ are compared as follows:

$$\delta(r, w) = \begin{cases} 1. & \neq_\delta & \text{if } \forall x \in \mathcal{N}\ x^{\langle r,w \rangle} = \neq_\mathrm{o} \\ 2. & =_\delta & \text{if } \forall x \in \mathcal{N}\ x^{\langle r,w \rangle} = =_\mathrm{o} \vee Q \\ 3. & >_\delta & \text{if } \forall x \in \mathcal{N}\ x^{\langle r,w \rangle} = >_\mathrm{o} \vee Q \\ 4. & <_\delta & \text{if } \forall x \in \mathcal{N}\ x^{\langle r,w \rangle} = <_\mathrm{o} \vee Q \\ 5. & ?{=}_\delta & \text{if } \exists x, y, z \in \mathcal{N} \text{ s.t. } x^{\langle r,w \rangle} = >_\mathrm{o} \wedge y^{\langle r,w \rangle} = <_\mathrm{o} \wedge z^{\langle r,w \rangle} = =_\mathrm{o} \\ 6. & ?_\delta & \text{if } \exists x, y \in \mathcal{N} \text{ s.t. } x^{\langle r,w \rangle} = >_\mathrm{o} \wedge y^{\langle r,w \rangle} = <_\mathrm{o} \end{cases}$$

where Q is the predicate $x^{\langle r,w \rangle} = \neq_\mathrm{o}$. The value of $\delta(r, w)$ is the lowest numbered case which applies.

The order notation has been used in the definition for clarity. In practice, standard diophantine equation analysis computes $\delta(r, w)$ [Nic84]. (Only one case holds for all r and w.) We frequently write $r >_\delta w$ for $\delta(r, w) = >_\delta$.

Locations are read and written in the same iteration if $r =_\delta w$. The relation $\neq_\delta$ is included for completeness; there is no dependency between two references if $r \neq_\delta w$. We assume that all such read/write pairs are eliminated from consideration. If $r <_\delta w$, then locations are read before being written. If $r >_\delta w$, then locations are written before being read. The relation $r\ ?_\delta\ w$ holds if and only if for some locations y and z, y is read before it is written $(r^{-1}(y) < w^{-1}(y))$ and z is written before it is read $(r^{-1}(z) > w^{-1}(z))$. In this case no *a priori* order of references can be established. The relation $r\ ?{=}_\delta\ w$ is included to distinguish the case where references can occur in either order and there is a location u where $r^{-1}(u) = w^{-1}(u)$.

This case analysis is similar to, but more refined than, the method of Wolfe [Wol78]. We have added the relations $?_\delta$ and $?{=}_\delta$. We use the more complex analysis because our methods apply when $r\ ?_\delta\ w$ or $r\ ?{=}_\delta\ w$.[2] The inclusion of $?_\delta$ and $?{=}_\delta$, however, greatly complicates the technical presentation. We omit these from most of what follows. Extensions to accommodate $?_\delta$ and $?{=}_\delta$ and an example are included in the appendix. The following definition extends the comparison of references to multi-dimensional arrays.

Definition 9.2.5 Let $A[r_1, \ldots, r_n]$ and $A[w_1, \ldots, w_n]$ be a read and write reference of A in L. Then

$$\delta(A[r_1, \ldots, r_n], A[w_1, \ldots, w_n]) = \langle \delta(r_1, w_1), \ldots, \delta(r_n, w_n) \rangle$$

This is a comparison vector.

For example, if the statement $A[x, y-4, z] := A[x+3, 2y+1, z]$ appears in L, the comparison vector for this read/write reference pair is $\langle <_\delta\ ?{=}_\delta\ =_\delta \rangle$.

Definition 9.2.6 Let $c = \langle c_1, \ldots, c_n \rangle$ be a comparison vector, where each c_i is $>_\delta$, $<_\delta$, or $=_\delta$. Let j be the least index such that c_j is not $=_\delta$. If there is such a j then c_k is a *defining position* for c if

$$c_k = c_j \wedge \forall z < k, c_z = \,=_\delta \vee c_z = c_j$$

[2] This is also why we prefer the term comparison vector to direction vector; there is no meaningful direction when an entry is $?_\delta$.

If there is no such j, then $n+1$ is taken to be c's only defining position.

A defining position of a comparison vector determines the lexicographic ordering of read and write references. For instance, let $c = \langle =_\delta <_\delta =_\delta <_\delta >_\delta \rangle$. The defining positions are c_2 and c_4; this comparison vector indicates that locations are read before being written. Intuitively, any legal quantization must preserve this order by not unrolling "too much" on dimension c_2 or c_4. This is formalized in the next section.

Definition 9.2.7 A comparison table of C for loop L contains a comparison vector

$$\delta(A_k[r_1,\ldots,r_n], A_k[w_1,\ldots,w_n])$$

for every distinct pair of read and write references of each array A_k. C_{ij} is the jth component of vector i.

As an example, consider Figure 9.1. There are two pairs of read and write references—$(A[i+1,j], A[i+1,j+1])$ and $(A[i+16,j], A[i+1,j+1])$. The comparison table for this loop is:

$$\begin{bmatrix} =_\delta & >_\delta \\ <_\delta & >_\delta \end{bmatrix}$$

9.2.2 Distance Tables

In this section we describe the *distance table*, which is used in conjunction with the comparison table to compute quantizations. For the computation of the unrolling of each loop, a notion of distance (in time) between references to the same array element is needed. A one-dimensional array B is used to introduce the idea.

Definition 9.2.8 Let $B[r]$ and $B[w]$ be read and write references of B in L using iteration variable i. A k-unrolling of loop l_i is *strict* if and only if

$$\forall x \in \mathcal{N}\ r^{-1}(x) \Box w^{-1}(x) \wedge Q \Rightarrow \left\lfloor \frac{r^{-1}(x)}{k} \right\rfloor \Box \left\lfloor \frac{w^{-1}(x)}{k} \right\rfloor$$

where $\Box$ is either $<$ or $>$ and Q is the predicate $r^{-1}(x) \in \mathcal{N} \wedge w^{-1}(x) \in \mathcal{N}$.

A quantization is strict if the execution of the quantized loop $\hat{L}$ does not contain dependent array references in an iteration of $\hat{L}$ that were not in the same iteration of L.

Definition 9.2.9 Let $B[r]$ and $B[w]$ be read and write references of B in L.

$$\rho(r,w) = \max_{z \in \mathcal{Z}^+ \cup \{\infty\}} \forall x \in \mathcal{N}\ r^{-1}(x) \Box w^{-1}(x) \wedge Q \Rightarrow \left\lfloor \frac{r^{-1}(x)}{z} \right\rfloor \Box \left\lfloor \frac{w^{-1}(x)}{z} \right\rfloor$$

where $\Box$ is either $<$ or $>$ and Q is the predicate $r^{-1}(z) \in \mathcal{N} \wedge w^{-1}(z) \in \mathcal{N}$.

Thus, $\rho(r,w)$ is the greatest strict unrolling of a loop with respect to r and w. Note that ∞ is a possible value. This is introduced to handle the case where there is no greatest strict unrolling—any unrolling up to and including full unrolling is strict. The value ∞ is greater than any value of $\mathcal{Z}^+$.

The value of $\rho(r,w)$ depends on $\delta(r,w)$. For example, if $\delta(r,w)$ is $=_\delta$, then $\rho(r,w) = \infty$. This follows because all dependent references occur in the same iteration if $r =_\delta w$. The value of $\rho(r,w)$ can be computed for the other cases of $\delta(r,w)$. Interestingly, it is possible to have strict unrollings greater than one even if $r\ ?_\delta\ w$ or $r\ ?{=}_\delta\ w$. The details are included in the appendix. We now define distance vectors and tables analogous to the comparison vectors and tables introduced earlier.

Definition 9.2.10 Let $A[r_1,\ldots,r_n]$ and $A[w_1,\ldots,w_n]$ be a read and write reference (respectively) of A in L. Then

$$\rho(A[r_1,\ldots,r_n], A[w_1,\ldots,w_n]) = \langle \rho(r_1,w_1),\ldots,\rho(r_n,w_n)\rangle$$

This is a distance vector.

Definition 9.2.11 A distance table D for loop L contains a distance vector

$$\delta(A_k[r_1,\ldots,r_n], A_k[w_1,\ldots,w_n])$$

for every distinct pair of read and write references of each array A_k. The order of the vectors in the table is assumed to be compatible with the comparison table for L. D_{ij} is the jth component of vector i.

As an example, consider once more Figure 9.1. The distance table for this loop is:

$$\begin{bmatrix} \infty & 1 \\ 15 & 1 \end{bmatrix}$$

9.2.3 Correspondence Between Tables and Loops

From the previous discussion, C and D can be computed for every loop L. In this section, we show that every *compatible* C and D pair corresponds to some loop L (compatibility is a simple well-formedness condition). This permits us to dispense with loops entirely and work only with the abstract representation.

Definition 9.2.12 Let C be a comparison table and D a distance table, with the elements of C restricted to $>_\delta$, $<_\delta$, and $=_\delta$. Then C and D are compatible if and only if

1. C and D have the same dimensions
2. $C_{ij} \in \{<_\delta, >_\delta\} \Rightarrow D_{ij} \in \mathcal{Z}^+$
3. $C_{ij} \in \{=_\delta\} \Rightarrow D_{ij} = \infty$

The following shows how to compute a compatible comparison and distance table from L.

Definition 9.2.13 Let L be a set of n nested loops. Then L^C, a comparison table, and L^D, a distance table, are defined for L:

$$L^C_{ij} = \delta(r^i_j, w^i_j)$$

$$L^D_{ij} = \rho(r^i_j, w^i_j)$$

where r^i_j is the jth component of the read reference of the ith read-write reference pair (w^i_j is similarly defined).

Theorem 9.2.14 For every compatible pair of comparison and distance tables C and D, there exists a loop L such that $L^C = C$ and $L^D = D$.

Proof: Construct n nested loops, where n is the number of columns of C. For each pair of vectors c_i of C and d_i of D construct a statement $R_i := W_i$, where R_i is a read reference and W_i a write reference of an array A_i such that $\delta(R_i, W_i) = c_i$ and $\rho(R_i, W_i) = d_i$. This can be done as each component of the vector is independent of the others and C and D are compatible. □

9.3 Computing Quantizations

In this section, we prove several results about the complexity of computing quantizations and develop an algorithm for computing *strict* quantizations. The order notation must be extended to work with comparison vectors rather than read/write reference pairs. We define x^c, where x is a location and c is a comparison vector for a read/write reference pair p, to be x^p. We write x^c_i for the ith component of x^c. The following key lemma justifies the change in notation.

Lemma 9.3.1 Let c and d be compatible comparison and distance vectors of L. For simplicity, we assume that c has no $?_\delta$ or $?{=}_\delta$ entries. If the quantization unrolls loop l_i more than d_i times, then there is a location x such that $x^c_i = =_0$ in $\hat{L}$. If the quantization unrolls loop l_i less than d_i times, then x^c_i in the $\hat{L}$ and x^c_i in L are always equal.

Proof: Follows immediately from the definition of a distance vector. □

We also make use of the fact that regardless of the unrolling of a loop l_i and comparison vector c, there is always some element x such that x^c_i is the same in both the quantized and original loops. This is true because we assume upper and lower loop bounds are arbitrary, and therefore loops may not be fully unrolled. As stated before, the technique can be extended to cover the case where loop bounds are known.

9.3.1 Properties of Loop Quantization

We now prove several general properties of loop quantization. The development is geared toward pinpointing exactly which quantizations are feasibly computable.

Definition 9.3.2 A quantization q of L is an n-tuple $\langle q_1, \ldots, q_n \rangle$, $q_i \in \mathcal{Z}^+ \cup \{\infty\}$, where $q_i = \infty$ indicates that loop l_i may be unrolled an arbitrary number of times.

Definition 9.3.3 Let q and q' be quantizations of L. Then $q \preceq q' \Leftrightarrow \forall j \; q_j \leq q'_j$ (i.e., the product ordering). We take ∞ to be greater than any element of $\mathcal{Z}^+$.

A quantization q is said to be *maximal* if it is a legal quantization and there is no quantization q' that is legal and $q \preceq q'$. A quantization q is *maximum* if it is legal and for any other legal quantization q', $q' \preceq q$. Clearly, for every L, there is a maximal quantization.

Lemma 9.3.4 There exists an L such that L has maximal quantizations q_1 and q_2 and $q_1 \not\preceq q_2$ and $q_2 \not\preceq q_1$.

Proof: Let $C = [>_\delta \; >_\delta \; <_\delta]$ and $U = [1\ 1\ 1]$. Then $\langle \infty \; 1 \; \infty \rangle$ and $\langle 1 \; \infty \; \infty \rangle$ are both maximal quantizations.

To see this, assume that $q_1 > 1$ and $q_2 > 1$. Applying Lemma 9.3.1, there is a location x such that $x^c = \langle =_o \, =_o \, <_o \rangle$ in the quantized loop. However, $x^c = \langle >_o \, >_o \, <_o \rangle$ in the original loop, showing that the quantization does not preserve the order of conflicting references. Also, if $q_1 = 1$ or $q_2 = 1$ then the order of references is preserved. Thus both quantizations are maximal. □

Barring the existence of a unique maximum, what should be the criteria for an optimal quantization? Ideally, we wish to maximize two things: the number of ∞ entries in the quantization and the product of the other components.

Maximizing the number of components that are ∞ permits the greatest leeway in code generation for an arbitrary machine and generally provides the greatest speedup (subject to the length of dependency chains induced by loop-carried dependencies). Let the *infinity number* $|q|_\infty$ of quantization q be the number of components that are ∞.

Definition 9.3.5 A quantization q of L is *best* if q is legal and

$$\forall q' \text{ s.t. } q' \text{ is legal, } |q|_\infty > |q'|_\infty \vee (|q|_\infty = |q'|_\infty \wedge \prod_{q_i \in \mathcal{Z}+} q_i \geq \prod_{q'_i \in \mathcal{Z}+} q'_i)$$

Theorem 9.3.6 Computing a best quantization of a set of nested loops L is NP-hard.

Proof: We actually prove a stronger statement: computing the infinity number of a best quantization is NP-hard. To help motivate the proof, consider the following tables:

$$L_C = \begin{bmatrix} >_\delta & >_\delta & <_\delta \end{bmatrix} L_D = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

Let $A[r_1, r_2, r_3]$ and $A[w_1, w_2, w_3]$ be read and write references which generate these tables. One of L_1 or L_2 must not be unrolled in any legal quantization. To see this, assume that L_1 and L_2 are unrolled k times, where $k > 1$. Then, by Lemma 9.3.1, there is a location x such that $x^{\langle r,w \rangle} = \langle =_o \ =_o \ <_o \rangle$ in the quantized loop and $x^{\langle r,w \rangle} = \langle >_o \ >_o \ <_o \rangle$ in the original loop. By unrolling more than once on L_1 and L_2 the quantization interchanges the order of references for at least one location.

The problem of computing the maximum infinity number is equivalent to selecting a minimum number of components of the quantization which must be set to some finite number. We can reduce Hitting Set [GJ79] to this problem.

Hitting Set: Given a finite set S and $T_1, \ldots, T_N$ subsets of S, what is the smallest cardinality of a set which contains at least one element of each of the T_i's.

Reduction: We construct a comparison table C of size $N \times |S| + 1$. Each row corresponds to a T_i and each column (except the last) to an element of S.

$$C_{ij} = \begin{cases} >_\delta & \text{if element } s_j \text{ is in set } T_i \\ <_\delta & \text{if } j = |S| + 1 \text{ (the last column)} \\ =_\delta & \text{otherwise} \end{cases}$$

We also need a distance table of the same size:

$$D_{ij} = \begin{cases} 1 & \text{if } C_{ij} \neq\, =_\delta \\ \infty & \text{otherwise} \end{cases}$$

Let

$$\begin{aligned} H &= \text{minimum hitting set cardinality of the } T_i\text{'s} \\ I &= \text{maximum infinity number of quantization for } C \text{ and } U \end{aligned}$$

Lemma 9.3.7 H = N - I

Proof: C looks like:

$$\begin{bmatrix} >_\delta & =_\delta & >_\delta & >_\delta & \cdots & <_\delta \\ =_\delta & =_\delta & >_\delta & =_\delta & \cdots & <_\delta \\ >_\delta & >_\delta & =_\delta & =_\delta & \cdots & <_\delta \\ \vdots & & & & & \end{bmatrix}$$

From the previous discussion, if a vector in C has $>_\delta$ entries at (say) positions i, j, and k, then a legal quantization must be strict in component i, j, or k. Thus a legal quantization corresponds in the obvious way to selecting a hitting set for the T_i's. Minimizing the size of the hitting set corresponds to maximizing the infinity number—in fact, the size of the minimum hitting set is the same as the number of components which must be set to a finite number in a best quantization. □

To complete the reduction, note that by Theorem 9.2.14 the quantization tables can be mapped to a loop in polynomial time. □

It is unfortunate that computing best quantizations is NP-hard. However, the exponential cost is proportional only to the number of dimensions of the largest array involved. If a loop contains references to n-dimensional arrays, a brute-force algorithm can compute a best quantization by performing 2^n simple tests. (The approach is to check every possible subset of the columns of D and C to find the one that optimizes $|q|_\infty$.) Because the number of array dimensions is small in practice, this may be a reasonable approach. We develop a polynomial time algorithm to compute a maximal strict quantization; however, computing a best quantization may be feasible in practice.

9.3.2 Strict Quantizations

A quantization of L is strict if the unrolled iterations of L in the loop body of $\hat{L}$ are data-independent. Following the line of development of the previous section, we say a quantization q is *maximal strict* if q is strict and there is no other strict quantization q' such that $q \preceq q'$. A quantization q is *maximum strict* if q is strict and for all other strict quantizations q', $q' \preceq q$. Finally, a quantization is *best strict* if it is strict and

$$\forall q' \text{ s.t. } q' \text{ is strict}, \ |q|_\infty > |q'|_\infty \vee (|q|_\infty = |q'|_\infty \wedge \prod_{q_i \in \mathcal{Z}+} q_i \geq \prod_{q'_i \in \mathcal{Z}+} q'_i)$$

These definitions parallel those from the previous section.

Theorem 9.3.8 The following are true:

1. For every L there is a maximal strict quantization.

2. There exists an L with no maximum strict quantization.

3. Computing the best strict quantization of L is NP-hard.

Proof: We develop an algorithm to compute a maximal strict quantization in the next section. Proofs of 2 and 3 mirror proofs from the previous section. □

The problems in this and the previous section stem from the difficulty of defining an "optimal" quantization. Three definitions of optimality were proposed for both the general and strict cases: maximal, maximum, and best. The structure of the two problems is identical. Maximal is computable in polynomial time, maximum does not always exist, and best is NP-hard. If a loop has a maximum quantization, then the following corollary shows that we can efficiently compute an optimal quantization.

Corollary 9.3.9 If a maximum quantization q exists for L then:

1. q is best
2. q is computable in polynomial time

Proof: One is immediate by the definitions of maximum and best. Two follows because a maximum quantization must be the unique maximal quantization and a maximal quantization is computable in polynomial time. □

9.3.3 Computing a Maximal Strict Quantization

A strict quantization is useful because the loops need not be unrolled to perform compaction; the unrolled iterations are always independent in a strict quantization. Thus a strictly quantized and compacted loop can be represented by the quantization vector and the (original) compacted loop body. Furthermore, for a single loop a strict quantization brings the loop into the form required for perfect pipelining. After unrolling, the loop-carried dependencies are between consecutive iterations. The algorithm also illustrates one application of the comparison and unrolling tables to quantization. A slightly modified algorithm computes a maximal quantization.

The algorithm consists of two passes over the tables. In the first pass a simple greedy heuristic computes an approximate quantization. The second pass refines this to a maximal strict quantization. We assume for technical simplicity that there are no $?_\delta$ or $?{=}_\delta$ entries in C.

The algorithm makes use of the fact that any strict quantization must preserve strictness in a defining position of every comparison vector. This is formalized in the following theorem.

Theorem 9.3.10 Let C and D be compatible comparison and distance tables of size $m \times n$. Let C_i be a vector in C. If C_i has a defining position less than $n+1$, then any strict quantization q must satisfy:

$$\exists k \text{ s.t. } C_{ik} \text{ is a defining position of } C_i \wedge q_k \leq D_{ik}$$

Proof: Straightforward application of Lemma 9.3.1 and the definition of a defining position. □

The strategy of pass one is as follows. Consider each column of C and D in order for $j = 1, \ldots, n$. Let

$$A_j = \{D_{ij} | C_{ij} \text{ is the leftmost defining position of vector } i\}$$

Then

$$\textbf{if } A_j \neq \emptyset \textbf{ then } q_j := \text{MIN}(A_j) \text{ else } q_j := \infty;$$

The idea behind this computation is simple. Let c be a vector in C. If c is defined at position k, then the relation c_k specifies the lexicographic ordering of references for the read/write reference pair which c represents. One way to preserve strictness is to unroll strictly on dimension k with respect to c. Note that strictness guarantees independence of the iterations. For example, let c be the vector $\langle =_\delta =_\delta <_\delta =_\delta >_\delta \rangle$. The comparison vector c is defined at c_3. By unrolling strictly with respect to c on the third dimension, we guarantee that conflicting references occur in the correct order (reads before writes) and in separate iterations. The first phase enforces strictness at the first defining position of each vector.

Minimizing over the set A_j gives an unrolling which satisfies the above requirements for all vectors. If $A_j = \emptyset$, then the column under consideration has no impact on the order of references (as determined by $q_1, \ldots, q_{j-1}$) and may be unrolled arbitrarily. Note that this can occur before the algorithm has passed a defining point of all vectors. For example, consider the comparison and distance tables:

$$C = \begin{bmatrix} <_\delta & =_\delta & =_\delta & >_\delta \\ =_\delta & =_\delta & <_\delta & =_\delta \\ >_\delta & >_\delta & =_\delta & =_\delta \end{bmatrix} D = \begin{bmatrix} 1 & \infty & \infty & 1 \\ \infty & \infty & 1 & \infty \\ 1 & 1 & \infty & \infty \end{bmatrix}$$

In this case, pass one generates $q = \langle 1\ \infty\ 1\ \infty \rangle$, although the second vector is not defined until column three. The reason is that because pass one unrolls strictly on column one, the correct order of reads and writes for vector three is already guaranteed. Column one is the *selected defining position* of vector three. Vectors one and two do not require any particular unrolling for column two; thus, $q_2 = \infty$.

Pass one does not, in general, compute a maximal strict quantization. A counterexample:

$$C = \begin{bmatrix} <_\delta & <_\delta & <_\delta \\ =_\delta & >_\delta & <_\delta \end{bmatrix} D = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

In this example, pass one computes $q = \langle 1\ 1\ \infty \rangle$; however, $q' = \langle \infty\ 1\ \infty \rangle$ is also strict and $q' \succ q$. The problem is that ordering and strictness for the first vector is satisfiable by unrolling strictly on any dimension. The second pass fixes this problem.

The second pass considers each column from right to left $(n, \ldots, 1)$. Fix a column j. Let J be the set of comparison vectors with a selected defining position less than j. For each $C_i \in J$, check if q_j, C_{ij}, and D_{ij} together guarantee the strictness of C_i—i.e., whether j is a defining position of C_i, and $D_{ij} \geq q_j$. If so, then let x be the current selected defining position of C_i. Set C_i's selected defining position to j and recompute q_x without considering C_{ix}. Note that q_j need not be recomputed—the unrolling of dimension j is already sufficient to guarantee the strictness of C_i.

The idea is that the first pass establishes strictness as "early" as possible—at the leftmost defining position. The second pass then moves the selected defining position as far right as possible while maintaining consistency with the quantization. In the example above, this yields $q = \langle \infty \ 1 \ \infty \rangle$, which is maximal (and maximum).

An algorithm to compute a maximal strict quantization is given in Figure 9.4. We assume without loss of generality that all vectors containing $\neq_\delta$ have been removed from C. We outline a proof of correctness for this algorithm.

Lemma 9.3.11 The algorithm of Figure 9.4 is correct.

Proof: Let q be the quantization computed by the algorithm. By construction this is a legal strict quantization. To show that it is maximal we prove that any $q' \succ q$ is not a legal strict quantization.

Let $q' \succ q$. The there is an j such that $q'_j > q_j$. Clearly $q_j < \infty$. Consider q_j after columns $n, \ldots, j+1$ have been processed by pass two. There is at least one vector C_i with the following property:

($*$) C_i's defining position is j and j is the least such position.

If C_i's selected defining position cannot be moved by the pass two examination of column j, then it follows that there is no defining position k of C_i greater than j that also has a strict unrolling for C_i (i.e., $q_k < D_{ik}$). Thus, by property ($*$) and Theorem 9.3.10, j is the only defining position of C_i that preserves strictness (and possibly legality, too). Choose C_i from all vectors satisfying these criteria so that U_{ij} is minimized. Then $q_j = D_{ij}$, and by Lemma 9.3.1 no larger unrolling is permissible. □

```
                 Input:  C   m × n comparison table
                         D   m × n distance table

declare array defined[1..m], unroll[1..n], q[1..n]
begin
    ∀i defined[i] := nil;
    (* Pass 1 *)
    for j = 1 to n do
        begin
            if ∀i(defined[i] ∨ C[i,j] = =_δ)
                then q[j] := ∞
                else q[j] := min_{i|defined[i]=nil} D[i,j];
            for i = 1 to n do if C[i,j] ∈ {<_δ, >_δ} ∧ defined[i] = nil
                    then defined[i] := j
        end;
    (* Pass 2 *)
    for j = n to 1 do
        for each vector C_i
            if j is a defining position of C_i and D[i,j] ≥ q[j] and defined[i] < j
                then
                    begin
                        x := defined[i];
                        defined[i] := j;
                        q[x] := min_{i|defined[i]=x} D[i,x]
                    end
end;
```

Figure 9.4: Algorithm for computing a maximal strict quantization.

Extensions for the relations $?_\delta$ and $?{=}_\delta$ are included in the appendix. Turning once more to Figure 9.1, the maximal strict quantization computed by the algorithm for this example is $\langle 15\ 1\rangle$.

A variation on this algorithm computes a maximal quantization. For a maximal quantization, the constraint is to preserve legality rather than strictness. For some comparison vectors, any quantization is legal. Referring to the comparison for the example of Figure 9.1, any quantization is legal for the vector $[=_\delta\ >_\delta]$. For any other vector, legality requires strictness, or else the order of references is interchanged for two elements. The second vector $[<_\delta\ >_\delta]$ of the comparison table for Figure 9.1 requires a strict quantization for legality. Discarding those vectors for which any quantization is legal and running the strict quantization algorithm on the resulting tables computes a maximal quantization. A maximal (and maximum) quantization for the example is $\langle 15\ \infty\rangle$.

9.4 Mitred Quantization

Loop quantization defines a partition of the iteration space, allowing parallelism to be exploited within each partition. Thus far, we have considered only rectangular quantizations, but other partitions of the iteration space are useful. In this section, we develop *mitred quantization*, which allows rhomboid partitions. We show that mitred quantization is more powerful than rectangular quantization. In particular, for nested loops where references are all *simple*, mitred quantization can always be used to produce arbitrarily large quantizations. A reference of the form $A[I + c]$, where I is an iteration variable and c is a constant, is a simple reference.

Mitred quantization can be viewed as rectangular quantization applied to a linear transformation of the iteration space. Thus, for loops where all indirect references are not simple, the results of the previous sections must be applied (in the transformed iteration space) to ensure the legality of mitred quantization.

For clarity, we assume throughout this section that all indirect references are simple.

So far, it has been impossible to arbitrarily quantize a loop containing a statement of the form:

$$\begin{array}{ll} (*) & A[i,j] \leftarrow f(A[i-k, j+k']); \\ \text{or} & A[i-k, j+k'] \leftarrow f(A[i,j]); \end{array}$$

The comparison tables for these statements are

$$C = \begin{bmatrix} <_\delta & >_\delta \end{bmatrix} D = \begin{bmatrix} k & k' \end{bmatrix}$$

and

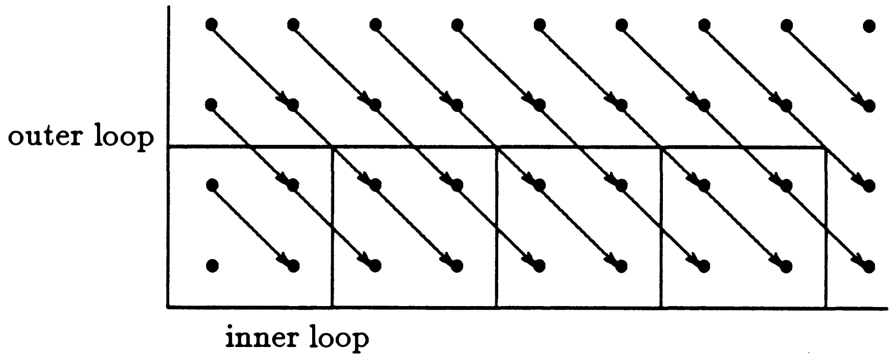
$$C = \begin{bmatrix} >_\delta & <_\delta \end{bmatrix} D = \begin{bmatrix} k & k' \end{bmatrix}$$

Any loop containing $(*)$ may be unrolled at most k times on loop L_i. For example, the outer loop in Figure 9.1 can be unrolled at most fourteen times. The problem is that the execution order of two dependent statements is reversed if L_i is unrolled "too much". However, if subscript expressions of the dependencies that prevent quantization are simple, then the corresponding dependent statements must lie on lines in the iteration space. In this case the quantization box can be skewed to include any dependent statements. An illustration is given in Figure 9.5. In the figure, the iteration space is depicted as a plane; each point is an iteration. The following lemma limits the cases we must consider.

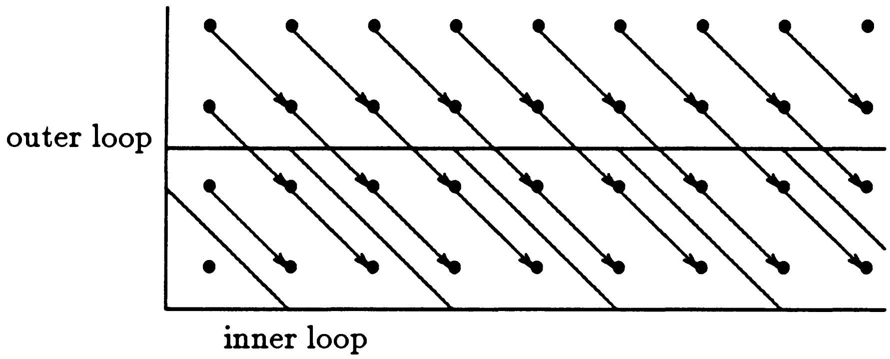
Lemma 9.4.1 Let C be a comparison table for loop L with simple references. Assume further that for any vector C_i of C, there is no k and l such that $C_{ik} = >_\delta$ and $C_{il} = <_\delta$. Then any quantization is legal for L.

Proof: Trivial. □

The lemma implies that the only cases of concern are of the type presented above. We limit the development to the two-dimensional case; we extend the



(a) Normal quantization "cuts" a dependency illegally.



(b) Skewed boxes capture dependent statements.

Figure 9.5: Quantization with simple references. Arrows are dependencies.

technique to arbitrary dimensions, but the essential ideas are best illustrated with examples that can be drawn in a plane. The mitred quantization algorithm generates arbitrarily large quantizations of any loop where the references preventing arbitrary quantization are simple. In particular, this allows arbitrarily large quantizations of any loop with only simple references.

Consider a loop L with a read reference $A[i+i_r, j+j_r]$ and a write reference $A[i+i_w, j+j_w]$. We assume that L consists of two nested loops, with i being the iteration variable of the outer loop. The *slope* of the dependency of a read/write reference pair is defined to be

$$\frac{i_r - i_w}{j_r - j_w}$$

If the slope of a dependency is negative, then the corresponding comparison vector must be $[<_\delta >_\delta]$ or $[>_\delta <_\delta]$. Let p be the read/write reference pair with the greatest negative slope s. It can be shown that our method ensures correctness for all dependencies if correctness is ensured for p.

Let $\Delta i{+}1$ and $\Delta j{+}1$ be unrollings of L_i and L_j respectively. The dependencies of p lie on lines of the form $i = sj + c$ for some constant c. With unrollings Δi and Δj the iteration $(\Delta i, \Delta j)$ must be executed after all points on the line $i - \Delta i = s(j - \Delta j)$ such that $0 \leq j < \Delta j$. These are exactly the dependent iterations that are executed before $(\Delta i, \Delta j)$ in normal loop execution order. The j-intercept of this line is

$$\frac{\Delta i}{-s} + \Delta j$$

To ensure that the intercept is an integer, we require that $\frac{\Delta i}{-s}$ be an integer. To execute iteration $(\Delta i, \Delta j)$, we must include in the quantization box, or have executed in an earlier box, all dependent statements on the line between the points $(\Delta i, \Delta j)$ and $(0, \frac{\Delta i}{-s} + \Delta j)$ We choose the limits of the quantization box to be the parallelogram bounded by the points $(\Delta i, 0)$, $(\Delta i, \Delta j)$, $(0, \frac{\Delta i}{-s})$, and $(0, \frac{\Delta i}{-s} + \Delta j)$. An illustration is given in Figure 9.6. (In the figure, the normal order of the axes is reversed to give a natural depiction of the iteration space.)

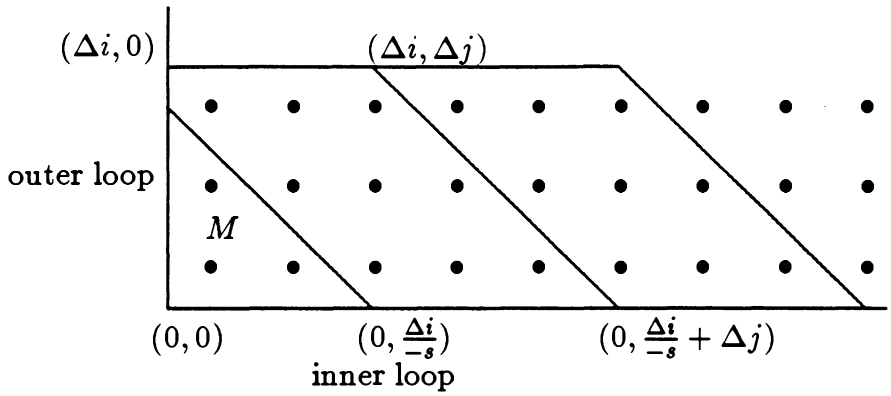


Figure 9.6: Computation of the mitred box for $\Delta i + 1$ and $\Delta j + 1$.

Note that all dependent references are included in the box or are satisfied by executing boxes in normal loop order (i.e., first along the j dimension, then the i dimension).

At the limits of the iteration space, some points are not included in any quantization box (area M in Figure 9.6). These iterations are executed, in normal loop order, as a prologue and epilogue to executions of the quantized loop $\hat{L}_j$. In general, a prologue for loop $\hat{L}_i$ is required, as $\Delta i + 1$ may not evenly divide the upper bound of loop L_i (see Section 9.5). A scheme for a mitred quantization $\langle \Delta i + 1, \Delta j + 1 \rangle$ is given in Figure 9.7. Note that there is an upper bound, computable at compile time, on the number iterations executed in the prologues and epilogues; thus these "extra" loops may themselves be quantized, up to and including full unrolling.

Lemma 9.4.2 Let L be two nested loops, and let $\hat{L}$ be a mitred quantization of L. Then L **sim** $\hat{L}$.

Proof: The proof is included in the appendix. □

```
(*
Original loop is
    for I = 0, N_I             loop L_I
        for J = 0, N_J         loop L_J
            B;
*)
for I = 0, N_I, Δi + 1
    begin
        (* L_J prologue—computes area M in Figure 9.6 *)
        for i = I, I + Δi
            for j = 1, Δi/(−s) − ⌊(i−I)/(−s)⌋ + ((N_J + 1) mod Δj) − 1
                B; where i replaces I, j replaces J
        (* The skewed quantization box. *)
        for J = Δi/(−s) + ((N_J + 1) mod Δj), N_J, Δj + 1
            begin
                B_00;
                B_01;
                ⋮
                B_0,Δj;          where    J + j − ⌊i/(−s)⌋ replaces J in B_ij
                B_10;                     I + i replaces I in B_ij
                ⋮
                B_Δi,Δj
            end;
        (* L_J epilogue *)
        for i = I + 1, I + Δi + 1
            for j = N_J − ⌊(i−I)/(−s)⌋, N_J
                B; where j replaces J, i replaces I
    end;
```

Figure 9.7: Scheme for arbitrary two-dimensional quantization $\Delta i + 1, \Delta j + 1$.

9.4.1 General Mitred Quantization

We motivate the generalization of the algorithm to n dimensions with an example. Let L be n nested loops, and let the comparison table for L be

$$\begin{bmatrix} <_\delta & =_\delta & >_\delta & \cdots & >_\delta \\ =_\delta & <_\delta & >_\delta & \cdots & >_\delta \end{bmatrix}$$

Consider the first vector. The defining position is column one, with a $<_\delta$ entry. Intuitively, every inner loop L_j with a $>_\delta$ entry must be skewed in the $(1, j)$ plane. However, the situation is complicated by the fact that some loops must be skewed with respect to more than one outer loop. In the second vector, the second column is a defining position, so the loop L_3 must be skewed with respect to loops L_1 and L_2. An extension of the proof of correctness for two-dimensional mitred quantization to n dimensions shows that a legal skew for a loop is the sum of the skews required by outer loops.

The general algorithm computes which loops must be skewed, and which loops must have unrollings that are multiples of a constant (recall that the outer loop must be unrolled by an amount divisible by the greatest negative slope of any dependency). This is given in Figure 9.8. The output of the algorithm is a smallest mitred quantization $\langle q_1, \ldots, q_n \rangle$ and a table *skew* where $skew[i, j]$ is the amount L_j must be skewed with respect to L_i. Figure 9.9 outlines the actual unrolling for an arbitrarily large mitred quantization of n loops. For clarity, the boundary conditions are omitted; the details are the same as in the two dimensional case.

We conclude this section with an example of mitred quantization of three nested loops. The original loop and the quantization tables are given in Figure 9.10a and 9.10b. (This loop is something of a canonical example for illustrating multiple-loop parallelization techniques [BCKT79,HL82,FM85].) A maximal normal (rectangular) quantization is $\langle 1\ 1\ \infty \rangle$. Mitred quantization can achieve arbitrarily large quantizations in all dimensions. Figure 9.10c shows a $2 \times 2 \times 1$

```
(* Input is nested loops L_1 ... L_n,
     and quantization tables C and D *)
let def[i] be a defining position of C_i;
(* skew[k, j] is set if L_j must be skewed with respect to L_k. *)
for each vector C_i do
     for j := 1 to n do
          begin
               s_j := max negative slope of dependency in plane (def[i], j);
               skew[def[i], j] := max(s_j, skew[def[i], j])
          end;
(* A loop must be unrolled a multiple of all the unrollings required
     by inner loops. *)
let q_i = lcm(skew[i, i + 1 ... n]);
```

Figure 9.8: Algorithm for n-dimensional mitred quantization.

mitred quantization.

9.5 The Details of Unrolling

Loop unrolling exposes many statements for scheduling by the core transformations. The parallelism available can be improved by simple transformations of the unrolled loop. For example, when the loop in Figure 9.11a is unrolled, the loop control code (the statements to increment the iteration variable and test for loop exit) must be duplicated in every unrolled iteration, as in Figure 9.11b. The control statements and the false dependencies they create are eliminated by two transformations. The first transformation deletes the increments to the iteration variable i and replaces every remaining occurrence of i in unrolled iteration c with $i + c$ [Ell85]. This produces the loop in Figure 9.11c. The intermediate exit tests are eliminated by ensuring that the original loop executes a number of iterations that is a multiple of the number of iterations in the unrolled loop body. If the loop unrolling is k, a prologue is added to the loop which executes

```
(* outline of algorithm to create n-dimensional mitred quantization
    ⟨..., Δi + 1, ...⟩ *)
for i := n to 1 (* for each loop ... *)
    begin
        let B be the current loop body;
        Replicate B Δi + 1 times:
            B_0;
            ⋮
            B_Δi;
        let I be the iteration variable of L_i;
        Replace "for I := 0 to N_i" with "for I := 0 to N_i by (Δi + 1)"
        for each L_j where skew[i, j] ≠ 0 do
            begin
                let J be the iteration variable of L_j;
                Replace J in B_h with J − ⌊h / −skew[i,j]⌋;
                Replace "for J := K to N_j" with
                    "for J := K + ⌊Δi / −skew[i,j]⌋ to N_j";
            end
    end;
```

Figure 9.9: An n-dimensional mitred quantization.

```
for i := 0 to N do
    for j := 0 to N do
        for k := 0 to N do
            begin
                A[i, j + 1, k] := B[i, j, k + 2];
                B[i + 1, j, k] := A[i, j, k + 6]
            end;
```

(a) Original loop.

$$C = \begin{bmatrix} =_\delta & >_\delta & <_\delta \\ >_\delta & =_\delta & <_\delta \end{bmatrix} \quad D = \begin{bmatrix} \infty & 1 & 6 \\ 1 & \infty & 2 \end{bmatrix}$$

(b) Comparison and unrolling tables.

```
for i := 0 to N by 2 do
    for j := 0 to N by 2 do
        for k := 9 to N do
            begin
                A[i, j + 1, k] := B[i, j, k + 2];
                B[i + 1, j, k] := A[i, j, k + 6];
                A[i, j + 2, k - 6] := B[i, j + 1, k - 4];
                B[i + 1, j + 1, k - 6] := A[i, j + 1, k];
                A[i + 1, j + 1, k - 2] := B[i + 1, j, k];
                B[i + 2, j, k - 2] := A[i + 1, j, k + 4];
                A[i + 1, j + 2, k - 8] := B[i + 1, j + 1, k - 6];
                B[i + 2, j + 1, k - 8] := A[i + 1, j + 1, k - 2]
            end;
```

(c) Loop with $2 \times 2 \times 1$ mitred quantization.

Figure 9.10: Mitred quantization of three loops.

the first N mod k iterations [CCK87]. The final program, with the intermediate increments and tests eliminated, is shown in Figure 9.11d.

This approach works well for Fortran-style DO loops, but does not cover more general forms of iteration. In a *while* loop, for example, the number of executed iterations is not known on entry to the loop, so intermediate exit tests cannot be removed. Compaction and unrolling apply regardless of the iteration construct, but other optimizations are dependent on the properties of a particular type of loop.

```
for i := 1 to N do
    A[i] := A[i] + 1;
```

(a) A sample loop.

```
for i := 1 to N by 3 do
    begin
        A[i] := A[i] + 1;
        if i = N then break;
        i := i + 1;
        A[i] := A[i] + 1;
        if i = N then break;
        i := i + 1;
        A[i] := A[i] + 1
    end;
```

(b) The loop unrolled.

```
for i := 1 to N by 3 do
    begin
        A[i] := A[i] + 1;
        if i = N then break;
        A[i + 1] := A[i + 1] + 1;
        if i + 1 = N then break;
        A[i + 2] := A[i + 2] + 1
    end;
```

(c) With increments removed.

```
j := N mod 3;
for i := 1 to j do
    A[i] := A[i] + 1;
for i := j + 1 to N by 3 do
    begin
        A[i] := A[i] + 1;
        A[i + 1] := A[i + 1] + 1;
        A[i + 2] := A[i + 2] + 1
    end;
```

(d) Without increments and tests.

Figure 9.11: Loop unrolling with optimizations.

Chapter 10

Compaction and the Wavefront Method

In this chapter we show how loop quantization and perfect pipelining can be combined to achieve the effect of a powerful high-level transformation, the *wavefront method.* The development illustrates the surprising expressive power of our compaction transformations and their relation to the wavefront method.

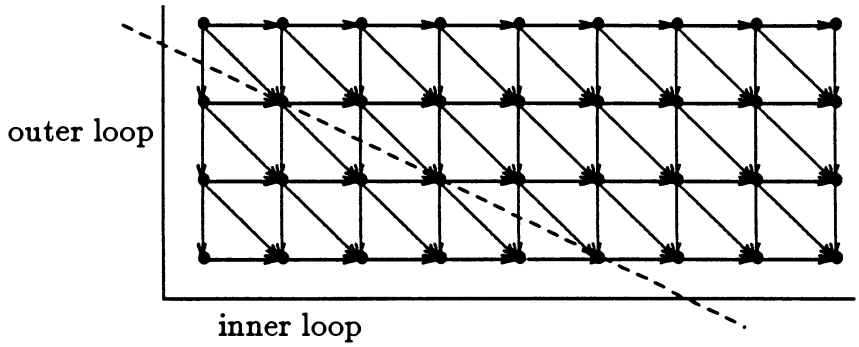
For simplicity, we restrict the development to two nested loops with a single assignment in the loop body. The results apply to arbitrary loops. Because of the simple flow-of-control in the example loops, we adopt a more readable representation than program graphs. Programs are written in a standard high-level syntax. All statements on each line of a program are executed in parallel; successive lines are executed sequentially.

10.1 The Wavefront Method

The wavefront method [Mur71,Lam74,Kuh80,Wol82,Wol87] extracts parallelism from multiple nested loops in many cases where parallelism cannot be found in any single loop. The traditional derivation of the wavefront method involves finding a legal wavefront through the iteration space. A legal wavefront preserves

for i := 1 **to** N_j **do**
  for j := 1 **to** N_i **do**
    $A[i,j] := A[i-1, j+1] + A[i+1, j] + A[i, j+1]$;

(a) A sample loop.



(b) The iteration space and optimal wavefront.

Figure 10.1: An example of the wavefront method.

dependencies; two dependent iterations are executed in the same order as in the original loop. All iterations on the same wavefront may be executed in parallel. When a loop is executed in the wavefront ordering, iteration $\langle i, j \rangle$ is executed at wavefront (time) step $i + j * b$, where the wavefront angle is $\arctan(-1/b)$.

The goal of the wavefront method is to find a wavefront that maximizes parallelism. Consider the loop in Figure 10.1a. Figure 10.1b shows the iteration space; the optimal wavefront angle is 30°. We use a slightly different formulation of the wavefront to make comparison with our technique direct. The wavefront is expressed as a line with slope $-1/b$ for some positive integer b. As in [Wol87], we do not consider the cases where the wavefront angle is 0° or 90°. (In these cases, the wavefront method does not apply.) There are many potential wavefronts that cannot be expressed by these ratios; however, the optimal wavefront is always of this form. In Figure 10.1b, the optimal wavefront slope is $-1/2$.

The wavefront must preserve dependencies; a wavefront that is too steep may reverse the order of dependent iterations. The following definition describes all legal wavefronts.

Definition 10.1.1 (Legal Wavefronts) Let L be two nested loops, let $-1/b$ be a wavefront slope, and let $\langle i,j\rangle$ and $\langle i',j'\rangle$ be the iteration vectors of any dependent iterations. Then the wavefront is legal if:

$$\langle i,j\rangle < \langle i',j'\rangle \Leftrightarrow i + j * b < i' + j' * b$$

10.2 Loop Quantization and the Wavefront Method

The following lemma shows the strong relationship between mitred quantization and the wavefront method.

Lemma 10.2.1 Let L be two nested loops. The following statements about L are equivalent:

1. The optimal wavefront slope is $-1/b$.
2. The greatest negative slope of any dependency is $-1/(b-1)$.
3. A mitred quantization box has slope $-1/(b-1)$.

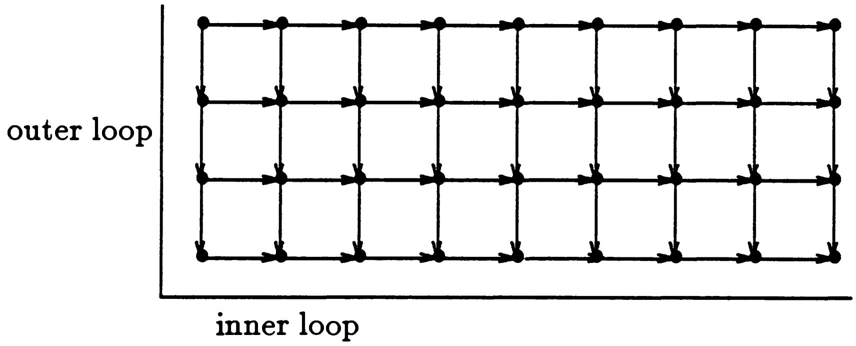
Proof: Let $1/(b-1)$ be the slope of the mitred quantization. This is also the greatest negative slope of any dependency. The wavefront slope $1/b$ is legal by Definition 10.1.1, because it preserves the dependency with the greatest negative slope. Furthermore, this is the best wavefront slope—a slope of $1/(b-1)$ would put two dependent iterations on the same wavefront. Finally, if $1/b$ is the best wavefront slope, then there must be a dependency of slope $-1/(b-1)$ and no dependency of greater negative slope. □

10.3 Loop Quantization and Perfect Pipelining

Loop quantization and compaction alone are insufficient to express the wavefront method. In principle, there is a fundamental difference. The wavefront method

```
for i := 1 to N_i
    for j := 1 to N_j
        A[i,j] := (A[i+1,j] + A[i,j+1] + A[i-1,j] + A[i,j-1])/4;
```

(a) Gauss-Seidel iteration.



(b) The iteration space.

Figure 10.2: Another loop.

expresses unbounded parallelism—the number of iterations that can be executed in parallel in the wavefront method is unrestricted, and parallelism grows with the size of the iteration space. In our compaction model, each node in the program graph can only contain a finite number of statements. However, this is not a meaningful difference; programs are executed on a machine with fixed resources—e.g., processors. We assume that there is a k such that no more than k iterations can be executed in parallel. We derive an algorithm that exposes at least as much parallelism as the wavefront method for any machine size k.

There is a more serious barrier to achieving the effect of the wavefront method using compaction. Consider the standard Gauss-Seidel iteration loop in Figure 10.2a; the iteration space with dependencies is shown in Figure 10.2b. Assume that the target machine has sufficient resources to execute three iterations of this loop in parallel. The loop unrolled with a three by four quantization is given in Figure 10.3. (We show below how to determine the amount of unrolling.) The loop after compaction—using the simple rule—is given in Figure 10.4. For

```
for i := 1 to Ni by 3 do
    for j := 1 to Nj by 4 do
        begin
            A[i,j] := (A[i+1,j] + A[i,j+1]+
                A[i-1,j] + A[i,j-1])/4;
            A[i,j+1] := (A[i+1,j+1] + A[i,j+2]+
                A[i-1,j+1] + A[i,j])/4;
            ⋮
            A[i,j+3] := (A[i+1,j+3] + A[i,j+4]+
                A[i-1,j+3] + A[i,j+2])/4;
            ⋮
            A[i+2,j+2] := (A[i+3,j+2] + A[i+2,j+3]+
                A[i+1,j+2] + A[i+2,j+1])/4;
            A[i+2,j+3] := (A[i+3,j+3] + A[i+2,j+4]+
                A[i+1,j+3] + A[i+2,j+2])/4
        end;
```

Figure 10.3: Loop with three by four quantization.

```
for i := 1 to Ni by 3 do
    for j := 1 to Nj by 4 do
        begin
            [i,j]
            [i,j+1]      [i+1,j]
            [i,j+2]      [i+1,j+1]  [i+2,j]
            [i,j+3]      [i+1,j+2]  [i+2,j+1]
            [i+1,j+3]    [i+2,j+2]
            [i+2,j+3]
        end;
```

Figure 10.4: Loop after quantization and compaction.

```
for i := 1 to N_i by 3 do
    for j := 1 to N_j by 6 do
        begin
            [i,j]
            [i,j+1]        [i+1,j]
            [i,j+2]        [i+1,j+1]    [i+2,j]
            [i,j+3]        [i+1,j+2]    [i+2,j+1]
            [i,j+4]        [i+1,j+3]    [i+2,j+2]
            [i,j+5]        [i+1,j+4]    [i+2,j+3]
            [i+1,j+5]      [i+2,j+4]
            [i+2,j+5]
        end;
```

Figure 10.5: Loop after further unrolling and compaction.

brevity, only the values of the iteration variables are given for each statement. Although full resource utilization is achieved in the middle of the loop body, utilization at the beginning and the end of the loop body is lower. Furthermore, regardless of the unrolling on either loop, after compaction there is always some start-up and wind-down code in the loop body.

The wavefront method also has start-up and wind-down periods where resources are less than fully utilized. However, for the wavefront method this occurs only once at the beginning and once at the end of the execution of the nested loops. Simply quantizing and compacting produces code that under-utilizes resources at the beginning and end of every quantized iteration.

We use perfect pipelining to circumvent this problem. Refer again to Figure 10.4. The loop has been compacted using the simple rule. Note that the third and fourth line of the compacted loop are the same except for increments to the iteration variable i. Figure 10.5 shows that further unrolling and compaction of the inner loop produces more parallel statements of the same type. Perfect pipelining infers this pattern from the loop in Figure 10.4; the resulting

```
for i := 1 to N_i by 3 do
    begin
        [i,1]
        [i,2]        [i+1,1]
        for j := 1 to N_j - 2 by 3
            begin
                [i,j+2]        [i+1,j+1]        [i+2,j]
            end;
        [i+1,N_j]        [j+2,N_j - 1]
        [i+2,N_j]
    end;
```

Figure 10.6: Loop after perfect pipelining is applied to the inner loop.

code is shown in Figure 10.6.

This loop is much better than the loop in Figure 10.4 and no more expensive to compute. However, it is still not as good as the wavefront method—the start-up and wind-down penalty is still paid for every iteration of the outer loop. Applying perfect pipelining to the outer loop eliminates this inefficiency. This is not quantization. The outer loop is treated as a single loop; the inner loop is treated as a single statement. The pattern overlaps the wind-down phase of one iteration of the j loop with the start-up phase of the next. The same procedure can be applied to any additional outer loops. The final code is shown in Figure 10.7. Figure 10.8 shows how this loop relates to the wavefront method. Each inner loop iteration executes "small" wavefronts. The end of each inner loop iteration is overlapped with the beginning of the next. The perfect dovetailing of the wind-down code for one iteration and the start-up code for the next is not accidental; this is a result of the regularity of the iteration space and always occurs.

We have glossed over many details in this example. In general, a non-trivial n-dimensional mitred quantization is required; this presents no additional problems.

$[1, 1]$
$[2, 1] \quad [1, 2]$
for $i := 1$ **to** $N_i - 3$ **by** 3 **do**
  begin
    for $j := 1$ **to** $N_j - 2$ **by** 3 **do**
      begin
        $[i, j + 2] \quad [i + 1, j + 1] \quad [i + 2, j]$
      end;
    $[i + 3, 1] \quad [i + 1, N_j] \quad [i + 2, N_j - 1]$
    $[i + 3, 2] \quad [i + 4, 1] \quad [i + 2, N_j]$
  end;
for $j := 1$ **to** $N_j - 2$ **by** 3 **do**
  begin
    $[N_i - 2, j + 2] \quad [N_i - 1, j + 1] \quad [N_i, j]$
  end;
$[N_j, N_i - 1] \quad [N_j - 1, N_i]$
$[N_j, N_i]$

Figure 10.7: Loop after perfect pipelining is applied to the outer loop.

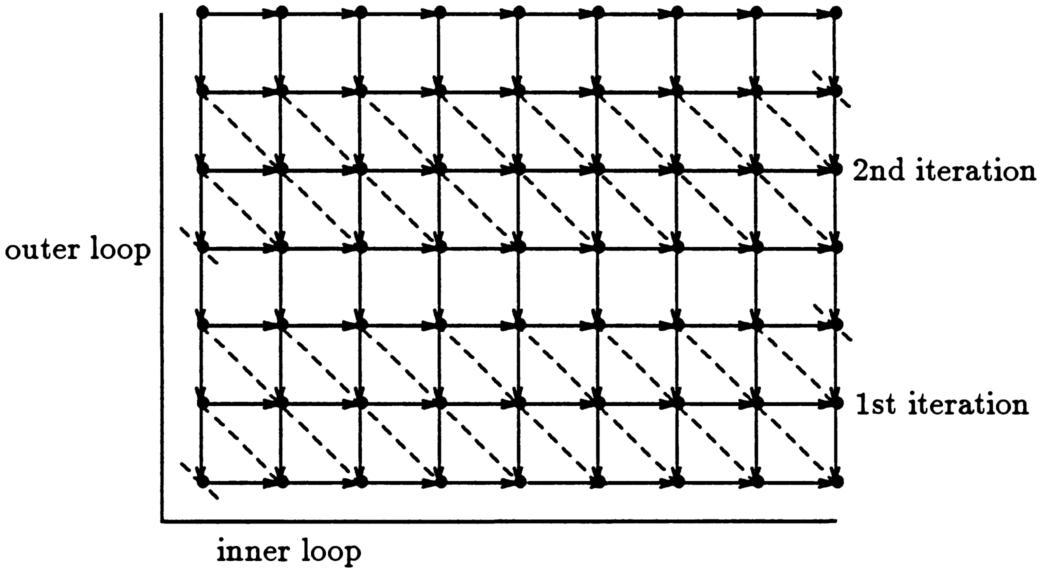


Figure 10.8: Execution of the final loop.

```
(* Given nested loops where mitred quantization is legal *)
Compute mitred quantization;
repeat
    increase unrolling of the outer loop;
    repeat
        increase unrolling of the inner loop;
        compact
    until pattern detected;
    if pattern utilizes resources sufficiently
        then break  (* leave repeat loop *)
until false;
Apply perfect pipelining to other loops;
Delete empty nodes;
```

Figure 10.9: A compaction algorithm for nested loops.

We have implicitly assumed that the loop limits of the final loop are divisible by three and that the iteration space is at least three by three. The techniques that compensate for these assumptions are discussed in Section 9.5.

10.4 The Algorithm

The algorithm is given in Figure 10.9. The inner loop is repeatedly unrolled and compacted until a pattern is detected. If the resulting pattern does not fully utilize the machine's resources, then the unrolling of the outer loop is increased and the process is repeated. The effect of this procedure is to generate larger and larger pieces of the wavefronts executed by the wavefront method. Because mitred quantization applies under exactly the same conditions as the wavefront method (Lemma 10.2.1), and because wavefronts can be arbitrarily large (limited only by the size of the iteration space), a large enough quantization combined with perfect pipelining fills the machine's resources and the algorithm terminates. Pipelining on the outer loops sustains utilization throughout execution of the nested loops.

10.5 Discussion

We have shown how the effect of the wavefront method can be achieved as a compaction transformation. Besides the theoretical interest, this has practical implications. It allows the power of the wavefront method to be used in compaction-based compilers. Although we considered only single statement recurrences in this chapter, the technique extends immediately to general loop bodies. In fact, on more complicated loops this algorithm performs better than the wavefront method because the underlying fine-grain compaction exploits parallelism between individual operations both inside and across iterations, and because quantization is legal in situations where the wavefront method is not.

Chapter 11

Extensions

Our model is an abstraction of real machines and real programs. In this chapter we outline extensions that translate programs in our abstract model to code for a real machine, and we show how our transformational system can be integrated with other program transformations. Our model of computation is closely related to the trace scheduling model. Most of the discussion in [Nic84,Ell85,CCD+87, CNO+87] on the validity of the model of computation also applies to our work.

This chapter is divided into three sections. The first section discusses how realistic machine resources are included in our model. The second section outlines a general scheme for using our system with transformations that alter the dependencies of a program. The extensions for handling several standard transformations are presented. Finally, we develop two detailed examples of loop parallelization using our system.

11.1 Modeling Machine Resources

As noted in Chapter 6, our model can be directly extended to handle resource constraints. A resource vector is associated with each node. A statement is not permitted to move into a node if it would violate the resource constraints. This approach is appealing because it preserves the uniformity of the model. However, there is a serious drawback. Resource conflicts between statements are a type of false dependency. If two statements x and y are data independent but both require the use of a critical resource, then they can be scheduled in either order (x before y or y before x) but not simultaneously. Thus, the incompleteness results of Chapter 4 become a serious problem—false dependencies caused by variable names can almost always be eliminated from a program, but false dependencies caused by resource constraints cannot be eliminated, and in fact must be common when scheduling for machines with few resources. Because the core transformations perform only local code motions, and scheduling with limited resources requires, in general, non-local transformations, extending the model to include resources in this way reduces the expressiveness of the system.

To circumvent this problem, we adopt a two-phase approach to handling machine resources. In the first phase, the compaction algorithms are applied without regard to resource constraints. In the second phase, the statements are heuristically rescheduled to accommodate the resources of the target machine. As shown in Chapter 8, it is possible (at least for machines with uniform resources) to prove upper bounds on the loss of performance due to rescheduling. Furthermore, experiments show that this two-phase approach results in little loss of performance in practice [Ell85].

A potential difficulty of this approach is that, in general, perfect pipelining requires resource constraints (Chapter 6). If there is no bound on the number of statements in a node, then perfect pipelining may not converge. For loops expressing a recurrence (i.e., a dependency cycle), a bound on the number of statements in a node arises naturally from the constraints imposed by dependencies. The optimal pipelining algorithm of Chapter 8 is a good example of this case. However, when there are no dependencies between the iterations of a loop (a so-called *doall* loop [PH79]), only resource constraints limit the number of statements in a node. The compact-and-reschedule approach can still be used in this case, by using resource limits considerably larger than those of the target machine during compaction, and then rescheduling. For machines with uniform resources, there is a direct algorithm within our formalism for generating optimal code for doall loops [NPA88].

11.2 Transformations that Alter Dependencies

The transformations we study preserve the data dependencies of a program. While this is a rich class of transformations, there are important transformations that alter the dependencies of a program. In this section, we outline how these transformations are intergrated into our system.

Arbitrary transformations can be applied to a program before or after compaction. A more interesting problem is how transformations outside of our system can be done incrementally during compaction. We present a general technique for integrating transformations with perfect pipelining. Renaming, induction variable removal, intermediate exit test removal, and common subexpression elimination can all be handled with this method. At the end of the section we present two examples of perfect pipelining with these other transformations.

Any transformation $\mathcal{T}$ can be used with perfect pipelining if it satisfies two conditions. The first requirement is that there is a mapping between statements in the original loop and statements in the transformed loop; that is, there are iterations $L'_0, \ldots, L'_{i-1}$ in $\mathcal{T}(\mathcal{C}u^iL)$ corresponding to iterations $L_1, \ldots, L_{i-1}$ in $\mathcal{C}u^iL$. Perfect pipelining converges when it discovers equivalent nodes under the mapping—i.e., two nodes whose counterparts under the mapping are equivalent. The second requirement is that given two nodes equivalent under the mapping, there is some additional transformation of the program that guarantees that the final loop is semantically equivalent to the original loop.

11.2.1 Renaming

Variable renaming is the most important transformation that aids parallelization. Without renaming to eliminate false dependencies, the effectiveness of parallelization is greatly reduced [Ell85,CKV85]. We consider the case of renaming scalar variables. Let L_j be an iteration in u^iL. For renaming to work with perfect pipelining, it must be done for all occurrences of a variable. Thus, if a definition of variable a is renamed in L_j, it must be renamed in all unrolled iterations. Let a_j be the new variable name for a in L_j. All uses of a corresponding to this definition must also be renamed a_j. A statement with a renamed variable corresponds to the original statement.

Let n and n' be the equivalent nodes detected by perfect pipelining. Node n executes statements from a set of iterations $L_i, \ldots, L_{i+k}$, while node n' executes statements from a set of iterations $L_j, \ldots, L_{j+k}$. Let p be a predecessor of n'. Create a new node, p', that is a successor of p and predecessor of n. Node p' contains statements $a_{i+h} := a_{j+h}$ for $0 \leq h \leq k$ to m. These statements "shift" the values of the variables so that the state at n is correct. The core transformations can overlap the statements of p' with the rest of the loop; thus, renaming of scalar variables does not necessarily introduce any extra steps in the loop.

11.2.2 Induction Variable Removal

A detailed description of induction variable removal is given in Section 9.5. Induction variable removal eliminates intermediate assignments to an induction variable in an unrolled loop [Ell85]. This transformation eliminates false dependencies caused by incrementing the induction variable. Let I be the induction variable of a loop L. To remove intermediate assignments to I, delete all occurrences of $I := I + 1$ and replace references to I in L_j with $I + j$. The modified statements correspond to the original statements; nothing (or a placeholder) corresponds to the deleted statements.

Let n and n' be the equivalent nodes detected by perfect pipelining. As above. node n executes statements from a set of iterations $L_i, \ldots, L_{i+k}$, while node n' executes statements from a set of iterations $L_j, \ldots, L_{j+k}$. Let p be a predecessor of n'. Create a new node p' that is a successor of p and a predecessor of n. Node p' contains the statement $I := I + (j - i)$ to p.

11.2.3 Intermediate Exit Test Removal

Intermediate exit test removal is also discussed in Section 9.5. Because it eliminates tests from a loop, this transformation both reduces the number of statements to be scheduled and code explosion. Let the exit test for L be "**if** $I \geq N$ **then goto** E". The transformation deletes each occurrence of this statement from $u^i L$.

Let n and n' be the equivalent nodes detected by perfect pipelining. Every cycle in the flow graph of $Cu^i L$ must have an exit test—otherwise, the program could diverge. However, different cycles in the pipelined loop may cover different numbers of iterations. (The loop in Figure 7.8 is an example of this situation.) Thus, when control reaches n in the pipelined loop, the number of iterations to be executed before the next exit test is unpredictable. Note, however, that if the loop is unrolled i times in performing perfect pipelining, then no cycle in the pipelined loop can cover more than i iterations. Let p be a predecessor of n'. Add the statement "**if** $I \geq N - i$ **then goto** E_n" to p. When the loop exits at this statement, there are less than i iterations left to execute. This epilogue code is computed and added at E_n.

Normally, the prologue and epilogue code are byproducts of perfect pipelining. Because some paths are deleted when exit tests are removed, however, perfect pipelining does not produce an adequate epilogue when used with this transformation. If the original loop has no tests other than exit tests, then the epilogue generated by perfect pipelining is the same as the epilogue required after intermediate exit tests are removed. An example of this case is given in Section 11.3.1.

11.2.4 Common Subexpression Elimination

Common subexpression elimination removes redundant computations from a program. When transformations that alter dependencies are used in conjunction with perfect pipelining, common subexpressions may be created. For example, induction variable removal creates common subexpressions. After induction variable removal, all occurrences of $I+c$ can be replaced by a variable T if a statement $T := I + c$ is added to the program.

For simplicity, we present a very general definition of common subexpression elimination. We assume that common subexpressions are statements that compute the same value (i.e., no common subexpressions arise within a statement). If statements $x_1, \ldots, x_n$ compute the same value, delete $x_2, \ldots, x_n$ and replace every use of the value written by x_i with a use of the value written by x_1. Except for the fact that statements are deleted, this is just variable renaming, so the same procedure applies.[1]

11.2.5 Tree-Height Reduction

Tree-height reduction exploits the associativity of an operation to transform a chain of dependent statements of length n into a balanced tree of dependent statements of depth $\log n$ [KM75]. For example, the sequence of statements in Figure 11.1 is evaluated in two steps instead of three after tree height-reduction is applied.[2] Tree-height reduction is important because many loops express recurrences built from an associative operation; thus tree-height reduction, which

[1]The same restrictions also apply; the common subexpressions must occur regularly throughout the iterations.

[2]In this example, tree-height reduction may change the semantics of the program because finite-precision arithmetic is not associative. If the numerical stability of a computation is greatly affected by reordering the sequence of arithmetic operations, tree-height reduction should not be used.

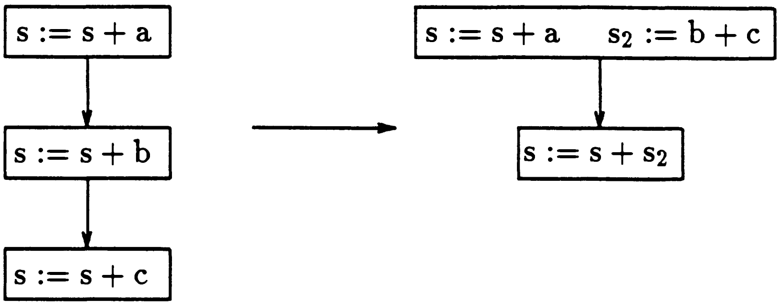


Figure 11.1: An example of tree-height reduction.

exploits semantic properties of operations, finds more parallelism than transformations based solely on dependency analysis.

Tree-height reduction is particularly powerful when combined with perfect pipelining. We briefly outline an algorithm for combining the two transformations for parallelizing loops with an associative recurrence. Apply tree-height reduction to the loop body and then apply perfect pipelining. If the resulting loop does not sufficiently utilize the target machine's resources, unroll the original loop, apply tree-height reduction to the unrolled loop body and apply perfect pipelining again. The parallelism tree-height reduction exposes increases with the unrolling of the loop body; thus, iterating this procedure is guaranteed to produce a loop which fills the machine's resources. (Note that this is similar to the algorithm for nested loop parallelization of Chapter 10.) We present an example of this technique in Section 11.3.2.

Tree-height reduction is an example of a transformation that does not fit the general scheme outlined at the beginning of this section. There is no correspondence between the statements in a loop before tree-height reduction and after tree-height reduction. Thus, there does not appear to be a general incremental algorithm for using tree-height reduction with perfect pipelining; in the iterative step of the algorithm outlined above, it appears necessary to begin again with the original loop body, unroll, apply tree-height reduction, and then apply perfect pipelining.

```
A[0] := 0;
A[1] := 1;
for i := 2 to n do
    A[i] := A[i − 1] + A[i − 2];
```

Figure 11.2: Fibonacci sequence example.

11.3 Examples

In this section, we develop two detailed examples. The first example illustrates how perfect pipelining is combined with transformations that alter dependencies. The second example illustrates perfect pipelining combined with tree height reduction, and shows how our techniques can be used to generate good code even for very resource-constrained machines.

11.3.1 Fibonacci Loop

The loop in Figure 11.2 computes the first n Fibonacci numbers. Figure 11.3 shows the unoptimized intermediate code. For this example, we assume that a load takes ten machine cycles, an integer add takes three cycles, and a register transfer takes one cycle. The only test in the loop body is an exit test; thus, the optimal scheduling algorithm of Chapter 8 is used.

As this loop is unrolled, many optimizations are applied. First, all intermediate exit tests (instances of statement A) are deleted. The intermediate assignments to the induction variable are also eliminated. All assignments to temporaries are given unique names. Common subexpression elimination deletes statements B and C from every iteration except the first; these values are computed by the increments to the induction variables. Common subexpression elimination also removes the loads D and E from every iteration except the first; these values reside in temporaries from previous iterations.[3] The loop body unrolled three times and optimized is shown in Figure 11.4. Statement labels establish

[3]This transformation is also called redundant load elimination [CCK87].

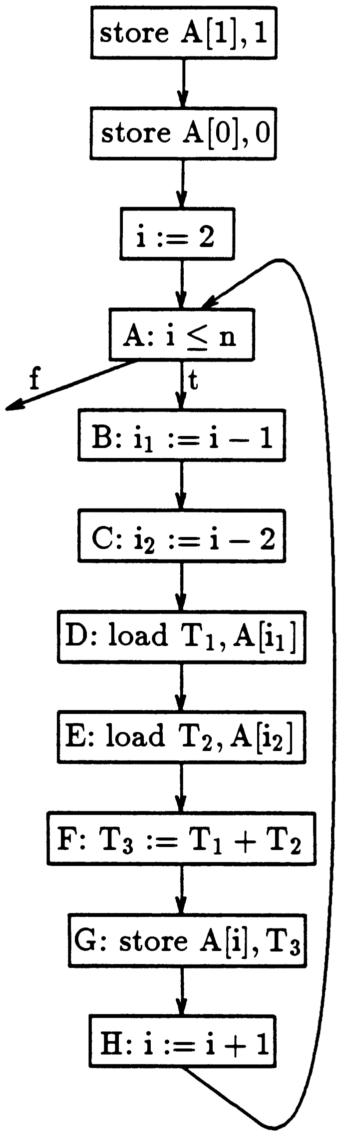


Figure 11.3: Intermediate code.

B_0: $i_1 := i - 1$;
C_0: $i_2 := i - 2$;
D_0: load $T_1, A[i_1]$;
E_0: load $T_2, A[i_2]$;
F_0: $T_3 := T_1 + T_2$;
G_0: store $A[i], T_3$;
H_0: $i_3 := i + 1$;
F_1: $T_4 := T_3 + T_1$;
G_1: store $A[i_3], T_4$;
H_1: $i_4 := i + 2$;
F_2: $T_5 := T_4 + T_3$;
G_2: store $A[i_4], T_5$;
H_2: $i_5 := i + 3$;

Figure 11.4: Loop unrolled three times and optimized.

the correspondence with original statements; subscripts indicate the unrolled iteration.

Applying the modified greedy scheduling algorithm (Figure 8.8) to this loop produces the schedule in Figure 11.5. Statement labels have been substituted for statements; the iteration of a statement is given by its column. In this example, greedy scheduling has been modified to account for instruction latencies. Thus, statement G (the store) is scheduled three cycles after statement F (the add) in each iteration. The pattern detected by perfect pipelining is shown in the boxed area of Figure 11.5.

The new loop after perfect pipelining and further optimization is shown in Figure 11.6.[4] The exit test and increment to the induction variable have been restored and moved to *n*. The new node, labeled *p*, contains the statements added to compensate for renaming. Many standard optimizations have been applied to the prologue code. Constant propagation eliminates the two loads of the first

[4]Perfect pipelining actually converges one iteration later than shown in Figure 11.6. This has no effect on the quality of the final code; we have presented this slightly simpler loop for clarity.

	iteration		
time	0	1	2
1	BCH		
2			
3			
4	DE		
5			
6			
7			
8			
9			
10			
11			
12			
13			
14	F		
15			
16		H	
17	G	F	
18			
19			H
20		G	F
21			
22			
23			G
24			
25			
26			

Figure 11.5: Fibonacci loop body after modified greedy scheduling.

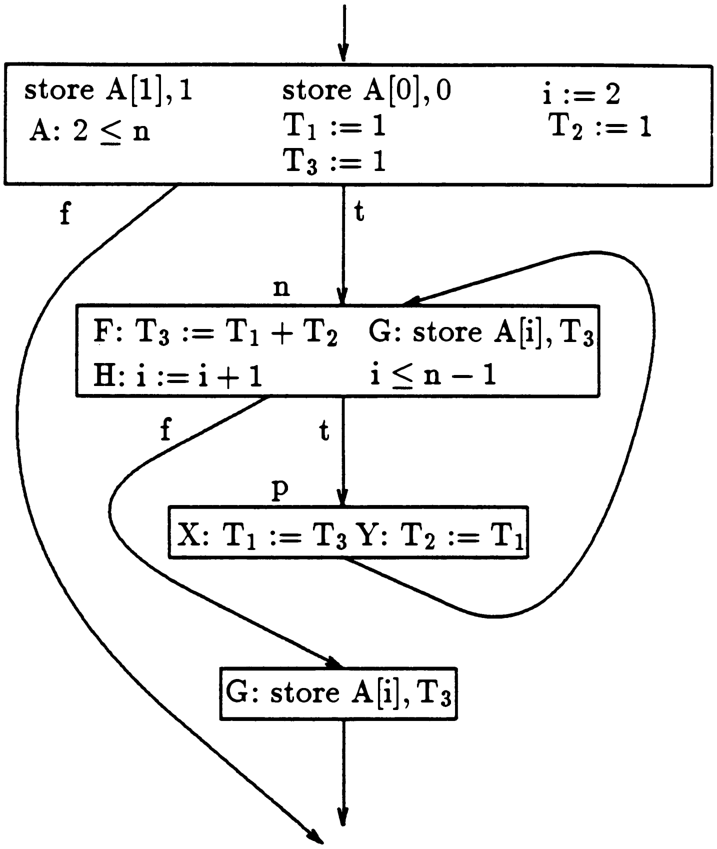


Figure 11.6: Loop after perfect pipelining.

iteration. The statements B_0 and C_0 are then deleted by dead code removal. Finally, the assignments to the temporary variables are simplified by constant propagation. Assuming sufficient resources are available, this loop requires four cycles per original loop iteration (three cycles for the integer adds plus one cycle for the update of the temporaries) with one cycle start-up and wind-down overhead.

This loop is further improved by the core transformations and standard optimizations. Statement Y in p is moved to n by *move-op*$(\{Y\}, p, n)$. Copy propagation eliminates statement X; node p is then empty and is deleted. All assignments to T_1 are deleted by dead code removal. The final code is shown in Figure 11.7. Assuming the target machine has sufficient resources, this loop requires three

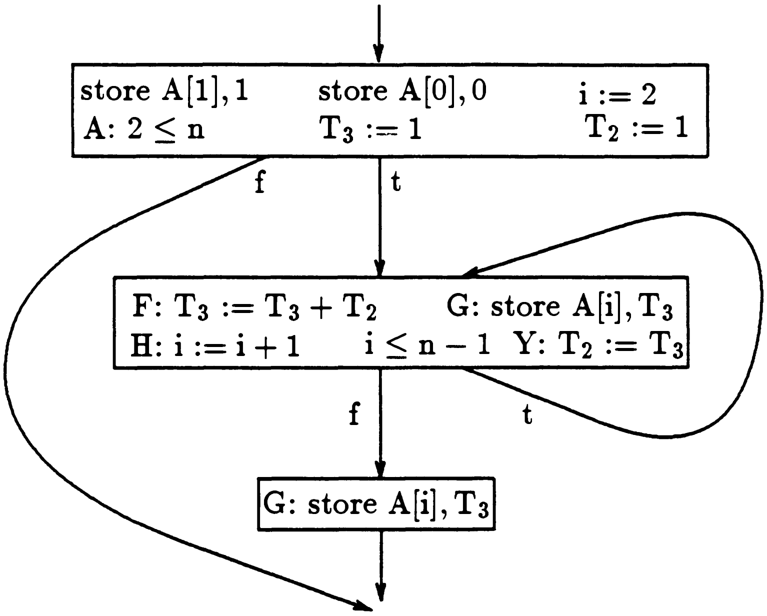


Figure 11.7: Final Fibonacci loop.

```
s := 0;
for i := 1 to n do
    s := s + A[i];
```

Figure 11.8: Summation loop.

cycles per original loop iteration.

11.3.2 Summation Loop

A loop that computes the sum of n numbers is given in Figure 11.8. We use this example to illustrate how perfect pipelining can be combined with tree height reduction, and to illustrate how our techniques apply in the presence of resource constraints. For this example, we assume that the target machine is a pipelined uniprocessor. We assume that a load takes ten cycles, floating point addition seven cycles, integer addition three cycles, a register transfer one cycle, and a test one cycle. The processor may initiate one statement every cycle.

The unoptimized intermediate code is shown in Figure 11.9. This loop takes

$$
\begin{array}{ll}
 & s := 0; \\
 & i := 1 \\
\text{loop:} & \text{load } T, A[i] \\
 & s := s + T \\
 & i := i + 1 \\
 & \textbf{if } i \leq n \textbf{ then } \text{loop;}
\end{array}
$$

Figure 11.9: Unoptimized intermediate code.

$$
\begin{array}{ll}
A_0: & \text{load } T_1, A[i]; \\
B_0: & s := s + T_1; \\
C_0: & i_2 := i + 1; \\
A_1: & \text{load } T_2, A[i_2]; \\
B_1: & s := s + T_2; \\
C_1: & i_3 := i + 2; \\
A_2: & \text{load } T_3, A[i_3]; \\
B_2: & s := s + T_3; \\
C_2: & i_4 := i + 3;
\end{array}
$$

Figure 11.10: Loop body unrolled three times and optimized.

21 cycles per iteration. The same loop unrolled three times and optimized is shown in Figure 11.10. In this case, only induction variable removal and intermediate exit test removal apply. The unrolled loop body after modified greedy scheduling is shown in Figure 11.11.

Perfect pipelining converges after three iterations are unrolled; cycles (nodes) 7 and 14 are equivalent. However, convergence at this point results in poor code because of an effect of the transformation to correct for renaming (Section 11.2.1). Note that the load issued in cycle 7 has not completed in cycle 14, but a statement will be added at that point to copy the result of the load into another variable for the next iteration. In the final code, this results in adding noops to the loop to allow the copy to take place after the load, with a corresponding loss

	iteration		
time	0	1	2
1	AC		
2			
3			
4			
5			
6			
7		AC	
8			
9			
10			
11	B		
12			
13			
14			AC
15			
16			
17			
18		B	
19			
20			
21			
22			
23			
24			
25			B

Figure 11.11: Summation loop body after modified greedy scheduling.

```
      load T1, A[1];
      load T2, A[2];
      s := 0;
      i := 2;
loop: i := i + 1;
      s := s + T1;
      load T1, A[i];
      if i > n - 1 then exit;
      i := i + 1;
      s := s + T2;
      load T2, A[i];
      if i ≤ n - 1 then loop;
exit:
```

Figure 11.12: Summation loop for a pipelined processor.

of performance. The solution to this problem is straightforward: disregard two equivalent nodes n and n' if values that must be copied at n' are not yet available. This guarantees that useful computation is scheduled until equivalent nodes n and n'' are found such that the copying can take place immediately at n''. (This is a general technique; convergence can always be postponed if it is desirable for some reason.) In this example, perfect pipelining converges after one additional iteration.

The pipelined loop is shown in Figure 11.12. The prologue and epilogue are omitted. After perfect pipelining, the statement "load T_2, A[i]" appears in the first node of the loop body. One core transformation moves this statement across the backedge to the end of the loop; this places it further from the point where T_2 is used. Rescheduling the resulting loop body produces the program in Figure 11.12.[5] This loop requires 6 cycles per iteration of the original loop. If the load into T_2 were not moved, then the loop would require 7 cycles per iteration

[5] A *list scheduling* heuristic is used [Fis79].

```
      (* Prologue ... *)
loop: A: i2 := i + 1;
      B: i3 := i + 2;
      C: i4 := i + 3;
      D: load T1, A[i];
      E: load T2, A[i2];
      F: load T3, A[i3];
      G: load T4, A[i4];
      H: s1 := T1 + T2;
      I: s2 := T3 + T4;
      J: s3 := s1 + s2;
      K: s := s + s3;
      L: i := i + 4;
      M: if i ≤ n − 4 then loop;
```

Figure 11.13: Summation loop with tree-height reduction.

of the original loop.

This loop is improved further using tree-height reduction. Figure 11.13 shows the result of tree-height reduction applied to the original loop unrolled four times. A modified greedy schedule of five iterations shown in Figure 11.14; as in previous examples, renaming, induction variable removal, intermediate exit test removal, and common subexpression elimination are applied.

The final code for this loop (after rescheduling the loop generated by perfect pipelining) is shown in Figure 11.15. This loop executes an original loop iteration every 3.1 cycles, a speedup of nearly 700% over the original loop. This loop is optimal; there are no empty cycles in the loop body. Tree-height reduction can expose arbitrary parallelism and perfect pipelining guarantees that the parallelism is uniformly exploited. Thus, for this loop—or any loop where tree-height reduction applies to a recurrence—the algorithm of Section 11.2.5 produces optimal code regardless of the latencies for loads and arithmetic operations.

	iteration				
time	0	1	2	3	4
1	ABCD				
2					
3					
4	EFG				
5					
6					
7					
8		C			
9					
10					
11		DEFG			
12					
13					
14	HI				
15			C		
16					
17					
18			DEFG		
19					
20					
21	J	HI			
22					
23				C	
24					
25					
26				DEFG	
27					
28	K	J	HI		
29					C
30					
31					
32					DEFG
33					
34					
35		K	J	HI	
36					

Figure 11.14: Summation loop with tree-height reduction and greedy scheduling.

```
      (* Prologue ... *)
loop: i := i + 4
      i2 := i2 + 4;
      i3 := i3 + 4;
      i4 := i4 + 4;
      load T1, A[i];
      load T2, A[i2];
      load T3, A[i3];
      load T4, A[i4];
      s3 := T5 + T6;
      s4 := T7 + T8;
      s1 := s2 + s3;
      s := s + s0;
      i := i + 4;
      i2 := i2 + 4;
      i3 := i3 + 4;
      i4 := i4 + 4;
      load T5, A[i];
      load T6, A[i2];
      load T7, A[i3];
      load T8, A[i4];
      s4 := T1 + T2;
      s5 := T3 + T4;
      s0 := s3 + s4;
      s := s + s1;
      if i ≤ n - 12 then loop;
      (* Epilogue ... *)
```

Figure 11.15: Final summation loop for a pipelined processor.

Chapter 12

Related and Future Work

In this chapter we survey other approaches to parallelization and discuss the relationship to our work. We also briefly outline the outstanding open problems in the area of compaction-based parallelization.

12.1 Related Work in Formalisms for Program Transformations

Although parallelization relies almost exclusively on transformations that exploit dependency information, no previous work has formally investigated the properties of these transformations. Allen did define the class of transformations based on dependency analysis in his thesis:

> A *reordering transformation* is any program transformation which merely changes the order of execution of the code, without adding or deleting any executions of any statements.[1]

In another paper, Allen and Kennedy note that although reordering transformations appear weak, the important parallelizing transformations are reordering transformations [AK84].

[1]From [All83], p. 18.

The style and approach of this work have been influenced by Paige's work in transformational programming [PK82,Pai83] and Guessarian's study of program transformation systems based on algebraic semantics [Gue79].

12.2 Related Work in Compaction

The core transformations were originally presented in [Nic85b] as the basis of percolation scheduling. The transformations presented here are somewhat different: move-op and unify are combined into a single transformation (also called move-op), and move-test is generalized to allow the movement of multiple tests in a single transformation.

Ebcioğlu has also designed compaction transformations based on percolation scheduling [Ebc87]. The transformations are similar to those used in this work, but the model of computation is considerably different. Ebcioğlu's model allows *conditional execution* of assignments. Instead of partitioning statements within a node into disjoint sets of assignments and tests, all statements are organized as a tree. Tests are branch points in the tree. As in our model, when a node is executed all tests are evaluated in the current store and the appropriate path is selected through the tree. An assignment is executed if it is on the selected path. While this model requires more hardware support than ours, it also permits shorter schedules.

Recently, Gupta and Soffa have developed a different approach to compaction [GS87]. Instead of working with control flow graphs, the transformations are defined on the control dependency graph of [FOW87].

12.3 Related Work in Software Pipelining

Software pipelining is not a new idea; programmers in the microcode community have done software pipelining by hand for decades [Kog77]. The first semi-automatic technique for software pipelining was proposed by Charlesworth

[Cha81]; an algorithm based Charlesworth's idea was implemented in a compiler for the FPS series [Tou84]. This and other early algorithms were limited to software pipelining loops without tests [RG81,SDJ84,SDX86]. Software pipelining has also been investigated in the context of compilation for dataflow machines [PA86].

In [FLS81], it is suggested that trace scheduling can be used for software pipelining. The idea is to simply apply trace scheduling and look for repeating states. No algorithm is presented; the idea is listed as a topic of current research. Lam and Ebcioğlu have independently observed that the pattern found by this method is likely to be unreasonably long [Lam87,Ebc87]. The results of Chapter 6 show that in general no pattern will be found at all, even if resources are limited—trace scheduling cannot be pipelined.

In 1987, four solutions to the problem of software pipelining loops with tests were published [Lam87,AN87b,Ebc87,SDX87]. Ebcioğlu designed a software pipelining technique and machine for scheduling systems (non-scientific) programs. Lam devised and implemented a software pipelining technique for Warp, a high-performance systolic array [AAG+86]. Su and his group proposed a software pipelining algorithm GURPR, based on an earlier software pipelining algorithm URPR [SDX86]. We discuss each of these approaches in some detail.

12.3.1 Ebcioğlu's Algorithm

Most closely related in spirit to our work is Ebcioğlu's algorithm [Ebc87]. The technique is applied to loop bodies that have been compacted using his variation of percolation scheduling. We briefly describe how the algorithm works. Two data structures are used: a partial schedule and a queue that contains *items*; an item represents an instruction that may be added to the schedule at some future point. At each step, one item is removed from the queue, an instruction (a node, in our terminology) is added to the partial schedule, and (possibly) some items are placed on the queue. The algorithm is non-backtracking. At each step, the partial schedule is extensible to a full schedule that is semantically equivalent to the original loop.

Initially, the queue is empty. Let n_0 be the root node of the compacted loop body. The instruction n_0 is placed in the schedule and for each $s \in succ(n)$, the item (s, n_0) is added the queue. The algorithm then repeatedly removes an item (n, p) from the queue. The node p is a composite of k nodes n_1 through n_k. Intuitively, this item represents a state in which the current iteration is expecting to execute n and the ith future (pipelined) iteration is expecting to execute n_i. If it is legal to overlap n with the k future iterations, a composite node consisting of n combined with p is added to the schedule, and an item (s, r) is added to the queue for every $s \in succ(n)$ and $r \in succ(p)$. If it is not legal to overlap n with the k future iterations, n is placed in the schedule by itself and an item (s, p) is added to the queue for every $s \in succ(n)$. Thus, if n cannot be executed in parallel with the overlapped iterations, the current iteration advances and the overlapped iterations wait. If an item placed on the queue corresponds to a node already in the schedule, the item is discarded.

Ebcioğlu shows that eventually the queue is empty and the algorithm terminates. We have omitted important details of the boundary conditions and how edges between nodes are updated in the partial schedule; a complete description of the technique is given in [Ebc87]. Several key characteristics of the algorithm are evident from our simplified description. First, statements appear in the same order in the pipelined loop body as in the original (compacted) loop body. Although delays may be added between statements due to dependency conflicts with other iterations or resource constraints, the order is fixed after compaction of the loop body. Second, the first statement of an iteration may not overlap the first statement of the previous iteration. Third, the algorithm encourages fast convergence by "reusing" pipeline states whenever possible. If the algorithm decides that n_1 can be overlapped with n_0 from the next iteration, then if n_1 is in an item n_0 will also be in the item. Composite nodes, once created, cannot be taken apart to allow other arrangements of statements. This reduces the number of equivalent nodes, speeding convergence.

The algorithm has several desirable features. The distance between the first and last statement of an iteration on a path of the pipelined loop body can be as much as $O(n^2)$ nodes, where n is the number of nodes in the original loop. Thus, dramatic reorganization of the code is possible with this technique. The algorithm is completely general; there are no awkward special cases or restrictions. Finally, while the worst case time complexity is exponential, the algorithm appears to run quickly in practice.

The algorithm also has drawbacks. It applies unevenly to all loops; while it performs extremely well on non-trivial examples, there are simple examples where it does not perform well. Consider the loop of Figure 7.4. Because the first statement of an iteration cannot overlap the first statement of the previous iteration, the algorithm performs no better than doacross on this loop (see Figure 7.5). Another example is the loop of Figure 8.5. The algorithm produces a schedule that executes an iteration every four steps, whereas the optimal loop schedule executes an iteration every three steps; this shows that the algorithm is sub-optimal even for loops without tests. The algorithm relies on the compacted loop body and composite nodes to guide the pipelining. The results of Chapter 8 show that the best organization of statements in the pipelined loop may have little to do with the best organization of statements visible in the first few iterations. Thus, fixing the pipeline states too early yields sub-optimal code.

It is not clear how Ebcioğlu's algorithm can be generalized to allow more flexible parallelization while retaining the very clever algorithmic structure. Of course, the simple rule can be trivially generalized to allow statements of an iteration to move anywhere within a fixed-size region of the program. For example, migrate could be applied to every statement, subject to the constraint that the distance between the first and last scheduled statement of any iteration on any path is at most a constant k. This is a bounded and local compaction algorithm, and therefore converges. If we choose $k = n^2$, where n is the number of statements in the loop body, this becomes a pipelining algorithm that performs the code motions of Ebcioğlu's algorithm without its restrictions. Unfortunately, this is probably impractical; an algorithm this general may not converge quickly enough in practice.

12.3.2 Lam's Algorithm

Lam has taken a completely different approach to software pipelining. Instead of focusing on data dependency constraints, Lam's algorithm is driven primarily by resource constraints. The basic technique, called *modulo scheduling*, has also been used in compilers for the FPS series [Tou84], the polycyclic machine [RG81], and Cydrome's Cydra [Cyd87].

Consider a loop L that requires a resource k times per iteration of the loop body. If the target machine has t of the resource, then an upper bound on the throughput is one iteration of L every k/t cycles. Let the *initiation interval* s be $\max(1, k/t)$. In modulo scheduling, the loop body is heuristically scheduled one statement at a time. When a statement is scheduled at time c, the instance of c in iteration i is scheduled at time $c+is$. If at any point a statement cannot be added to the schedule due to resource and dependency constraints, then the schedule is abandoned and the algorithm either backtracks or tries a larger initiation interval.

Lam develops a non-backtracking variation on this approach for the Warp architecture. The core of the compiler is a routine that schedules a code fragment with an acyclic dependency graph. A standard *list scheduling* heuristic [Fis79] is used, with some modifications to enhance software pipelining. To handle loops with cyclic dependency graphs, a different set of heuristics is used to first schedule the strongly connected components of the dependency graph. If a schedule cannot be found for a given initiation interval, the initiation interval is increased by one. The process is repeated until either a schedule is found or the initiation interval equals the length of the original loop body.

Lam extends modulo scheduling to loops with tests using *hierarchical reduction.* Consider an if-then-else construct in a standard high-level language. With hierarchical reduction, the then and else branches of a test are first scheduled independently. The shorter branch is padded with noops to make it the same length as the longer branch, and the scheduler encapsulates the entire if-then-else construct as a single statement. The resource requirements and dependencies of the new node are the union of the resource requirements and dependencies of the two branches. The new node hides flow of control within it, and thus can be treated like any node in a basic block by the scheduler. When a pipelined schedule is found, encapsulating nodes are expanded, recursively, to generate a flow graph of the program.

Without hierarchical reduction, modulo scheduling does not apply to loops with tests. Lam's work represents a significant advance in this respect. There are, however, two drawbacks to hierarchical reduction. First, information is lost when statements are encapsulated. Second, the fact that the hierarchical structure of the original program must be preserved limits possible code motions. For example, consider again the loop in Figure 7.4. A pipelining technique using hierarchical reduction degenerates to doacross in this case both because the dependencies between appends must be enforced on all branches in the encapsulated node, and because a statement cannot be scheduled in different places on different paths.

Lam has performed extensive experiments with her technique on the Warp machine. The results are very good; most loops in the experiment are scheduled optimally. The speedups achieved are modest (a factor of three is typical) because

Warp is very resource-constrained. It is not clear that these results scale to larger machines. For a machine with many resources, the cumulative effect of hierarchical reduction's overestimation of resource requirements and restrictions on code motion could be significant. There is no such impact when scheduling for Warp because there are not sufficient resources to overlap many statements. Referring again to Figure 7.4, Lam's algorithm schedules the loop optimally (or nearly so, depending on instruction latencies) for a machine that can perform one test and one memory write in parallel, but it also generates the same parallel code for any larger machine.

12.3.3 GURPR

GURPR, for Global Unrolling, Pipelining, and Rerolling, is a software pipelining technique proposed by Su, Ding, and Xia [SDX87]. The technique is based on URPR, an algorithm for pipelining loops without tests similar to doacross [SDX86]. Given a loop L, the first step of GURPR is to apply URPR to each path through the original loop body. The separate pipelined paths are then put together to form the pipelined loop, with compensation code added at points where execution could jump from one path to another.

The approach is similar in philosophy to trace scheduling—paths are first optimized as basic blocks, ignoring jumps into and out of the path, and then fix-up code is added to ensure correctness. GURPR is also subject to the same criticism as trace scheduling. There is no reason why the execution of a program should repeatedly follow the same path through the loop body. We feel that percolation scheduling provides a more uniform approach, overlapping iterations on all paths, rather than a subset of paths.

12.4 Related Work in Parallelization of Nested Loops

The ideas presented here on multiple loop unrolling have undergone considerable evolution. Loop quantization was proposed by Nicolau [Nic85a]. The formalism and algorithms developed in Chapter 9 were first presented in [AN87a].

Nested loop parallelization has been a major focus of attention of researchers investigating parallelization for vector and multiprocessor machines. The first algorithm, the wavefront method (or hyperplane method) [Mur71,Lam74,Kuh80, Wol82,Wol87], was a vectorization technique. More recently, multiprocessor parallelization has spurred interest in the generalization of doacross and doall loops to multiple nested loops [PH79,Cyt84,Cyt86,CP86,Pol87]. (A doall loop is the special case of doacross loops with zero delay between iterations.)

Cytron presents a multiple-loop doacross algorithm in [Cyt86]. This is essentially a cross between the wavefront method and the standard single loop doacross algorithm. The algorithm computes a delay d_i for every nested loop L_i. For n nested loops, iteration $\langle j_1, \ldots, j_{i-1}, j_i + 1, j_{i+1}, \ldots j_n \rangle$ must not begin execution until d_i steps after iteration $\langle j_1, \ldots, j_i, \ldots, j_n \rangle$. The effect of this method is achieved by the nested loop pipelining algorithm of Chapter 10 with the simple rule.

A standard transformation for multiprocessor parallelization and vectorization is *loop interchange* [Wol82,AK84,Wol86]. Loop interchange alters the order of nested loops. A simple example of loop interchange is given in Figure 12.1. The following lemma establishes the correspondence between loop interchange and loop quantization.

Lemma 12.4.1 Let L have n nested loops. Consider a quantization q where $q_i = \infty$ and all other components are 1. It is legal to interchange loops L_i and L_n if and only if q is a legal quantization of L.

```
for i := 1 to N do
      for j := 1 to N do
            A[i, j] := B[i, j] + B[i, j];
```
(a) Original program.

```
for j := 1 to N do
      for i := 1 to N do
            A[i, j] := B[i, j] + B[i, j];
```
(b) Program with loops interchanged.

Figure 12.1: An example of loop interchange.

Proof: Fully unrolling loop L_i and not unrolling any other loop has the same effect as interchanging L_n and L_i. □

Lemma 12.4.1 can be used to test whether two loops can be interchanged. Consider two loops L_i and L_j, where $i < j$. Restrict the quantization tables to loops L_1 through L_j. The loop L_j is now the innermost loop of the restricted table and the test of Lemma 12.4.1 applies. Lemma 12.4.1 also shows that loop interchange is a dependency-preserving transformation, or a *reordering* transformation in the terminology of [All83]. Let *LI* stand for loop interchange. By Lemma 12.4.1, P **sim** $LI(P)$.

It is not surprising that the quantization tables can be used to test for loop interchange, because both the conditions presented in [AK84] for valid loop interchange and loop quantization are based on direction vectors [Wol78]. More interesting is that our transformational system can implement loop interchange. Loop interchange transposes the iteration space of the two loops that are affected. This is not possible with loop quantization alone because loop quantization unrolls a loop a finite number of times. Therefore, loop quantization can only affect the order of execution of the iterations inside the quantization "box"; the quantized iteration space is still executed in normal loop order. However, loop quantization approximates loop interchange, and by taking limits with perfect pipelining the full effect of loop interchange can be realized. Allen has shown that arbitrary interchanges can be expressed as a series of interchanges between adjacent loops [All83]. To interchange adjacent loops L_i and L_{i+1}, where L_{i+1} is the inner loop, treat the body of the inner loop L_{i+1} as a single statement. Quantize this loop body on the outer loop L_i and perform no compaction. No compaction is a bounded and local compaction algorithm, so perfect pipelining applies. The pattern detected is the original loop body, but with iteration on loop L_{i+1} interchanged with iteration on loop L_i. Of course, substituting a more interesting compaction algorithm would achieve some parallelization as well as the effect of loop interchange.[2]

In [CCK87], Callahan, Cocke, and Kennedy propose another nested loop parallelization technique, *unroll-and-jam.* Figure 12.2 illustrates the idea. The outer loop is unrolled, creating multiple copies of the inner loop; in this case there are two copies. An attempt is made to combine the copies of the inner loop using *loop jamming.* (Loop jamming is a standard compiler optimization for sequential machines [AC72].) No analysis is presented in the paper; however, it is clear that this technique achieves exactly rectangular quantizations.

In [Wol87] Wolfe presents a technique, *loop skewing*, for altering the iteration space of a loop to improve vectorization. The idea is to skew the iteration space from a rectangular to a rhomboid shape. Wolfe's technique leads to a clean derivation of the wavefront method for vector machines. There is a close connection with the idea behind mitred quantization, where the quantization box is skewed to capture dependent statements. We believe that loop skewing combined with unroll-and-jam could have the same expressive power as general mitred quantization.

In [Wol86], Wolfe generalizes loop interchange to include triangular loops and imperfectly nested loops. The results apply immediately to loop quantization,

[2] We are also not suggesting that loop interchange should be implemented in this way; the purpose is to show that our system has the expressive power of loop interchange.

```
for j := 1 to M do
    for i := 1 to N do
        begin
            A[i, j] := A[i + 1, j] + A[i - 1, j];
            B[i, j] := A[i, j - 1] + B[i - 1, j]
        end;
```

(a) The original loops.

```
for j := 1 to M by 2 do
    begin
        for i := 1 to N do
            begin
                A[i, j] := A[i + 1, j] + A[i - 1, j];
                B[i, j] := A[i, j - 1] + B[i - 1, j]
            end;
        for i := 1 to N do
            begin
                A[i, j + 1] := A[i + 1, j + 1] + A[i - 1, j + 1];
                B[i, j + 1] := A[i, j] + B[i - 1, j + 1]
            end
    end;
```

(b) Loops after normal (single loop) unrolling on the outer loop.

```
for j := 1 to M by 2 do
    for i := 1 to N do
        begin
            A[i, j] := A[i + 1, j] + A[i - 1, j];
            B[i, j] := A[i, j - 1] + B[i - 1, j];
            A[i, j + 1] := A[i + 1, j + 1] + A[i - 1, j + 1];
            B[i, j + 1] := A[i, j] + B[i - 1, j + 1]
        end;
```

(c) Loops after jamming.

Figure 12.2: Unroll-and-jam example from [CCK87].

allowing the quantization of these loops as well.

12.5 Future Work

Considerable work remains to be done in the area of compaction-based parallelization. The major open question is whether there is an algorithm to compute optimal parallel schedules for loops with tests. The results of Chapter 8 suggest that there may be such an algorithm. However, the analysis used to prove the optimality of the algorithm for loops without tests does not directly apply. The problem is that the same definition of optimality cannot be used for loops with tests. The results of Chapter 4 show that in general some paths must be optimized at the expense of others. Thus, any reasonable definition of optimality must be with respect to priorities assigned to the paths of a loop. Given a definition of optimality, it is not clear that an optimal algorithm exists, or, if it does exist, that it is practical. (Because the simple rule has exponential time and space complexity, any optimal algorithm must also have exponential time and space complexity.) Even if an optimal algorithm were too expensive to be practical, it would be useful as a tool for analyzing the merits of practical alternatives.

Another outstanding problem is how to deal with code explosion. Every reasonably powerful compaction-based system suffers from worst-case exponential code explosion. This must be the case because a tightly-coupled machine has a single flow of control, and a program overlapping n independent computations, each with k paths, has k^n possible outcomes. This problem has probably not been addressed in depth in the past because it is unclear how code explosion can be controlled with most techniques. Consider trace scheduling. It is difficult to estimate beforehand how much code explosion will result from compacting a trace. Because copying is intertwined with the compaction of a path, proposed heuristics for restricting compaction to reduce code explosion in trace scheduling are relatively coarse [LA82,Ell85].

A more robust approach may be possible using percolation scheduling. Each core transformation can add only a small amount to the size of the program. Therefore, code explosion can be closely monitored during compaction—there is no danger that a single transformation will dramatically increase the size of the program. If code explosion becomes too severe, the compaction strategy can be modified to restrict code motion. Because the core transformations are very fine, the trade-off between space (the size of the final code) and time (the performance of the final code) can be precisely controlled—only the performance that must be sacrificed to keep the code size manageable is actually lost. Thus, using percolation scheduling it may be possible to design compaction algorithms which dynamically adjust to the changing control and dependency structures of a program.

Chapter 13

Conclusions

This work began as a project to build an interactive environment for parallelization [AN88a]. ESP (an Environment for Scientific Programming) has been operational since 1986. The core transformations, the algorithms of Chapter 5, and a simple version of loop quantization are implemented, as are many standard compiler optimizations. Interaction with the environment is through the program graph. In a typical action, the user selects a statement in a node and invokes a transformation from a menu; the compiler performs the transformation and redisplays the program graph. The complete hierarchy of transformations is available to the user—any transformation from the lowest-level core transformation to global compaction algorithms can be applied under user control. Details of the environment and the results of early experiments can be found in [AN88a].

Our work on software pipelining and the formalism for parallelization developed after the environment was completed. The environment itself contributed a great deal to this later work—actually being able to see what the algorithms did helped develop intuition about the behavior of the transformations in a way that analysis could not. Currently, the theory is ahead of the implementation. While the algorithm of Chapter 8 is implemented, general perfect pipelining is not. Also, relatively minor aspects of the environment which seemed appropriate when it was implemented are now either obviously inappropriate or prevent a clean implementation of the new techniques. Work is currently underway to adapt the system to the new techniques. Conclusive experimental evidence of the viability of our approach must await the new implementation.

13.1 An Evaluation

In Section 1.1, we listed properties of a good program transformation system. These properties are correctness, applicability, improvement, expressiveness, and efficiency. We summarize our results by evaluating our system by these criteria.

A program transformation is correct if it preserves program semantics. We have shown that the core transformations, perfect pipelining, and loop quantization are correct. All of the algorithms we have presented are composed from these transformations; therefore, these algorithms are also correct. Our transformations have very good applicability—each transformation involves only easily computed, local changes to a program.

Our formal measure of improvement is $\leq_p$. If $P \leq_p P'$, then P' is a better—more parallel—program than P. Each of the core transformations improves a program by this measure. The relation $\leq_p$ also serves as the basis for formalizing expressiveness. We defined completeness of parallelizing transformations, and showed that the core transformations are complete for a subset of all programs. The strongest evidence for the expressiveness of our system, though, is that the effect of many important parallelizing transformations for different machine models can be achieved with our system. There are two interpretations of these results. First, we have established a foundation for parallelization, encompassing transformations as diverse as trace scheduling and the wavefront method. Second, we have shown that effect of these techniques can be realized as special cases of a parallelization system based on the core transformations.

In Section 1.1, we observed that trade-offs amongst the criteria are necessary in the design of a parallelizing compiler. Efficiency is the criterion where we have compromised—the worst-case running time of any powerful compaction-based system is exponential. We are confident, however, that the system we have presented is practical. At the moment, the most convincing evidence for this claim is the existence of a successful commercial VLIW machine, Multiflow's TRACE, with a compaction-based compiler. Compilation speed for the TRACE, while slow, is reasonable [Rut88]. We expect that a full compiler based on our approach will be substantially faster, as well as more powerful, than trace scheduling, because perfect pipelining produces better results with much less unrolling.[1]

13.2 Compaction versus Vectorization

To place our work in perspective, we use our criteria to analyze parallelization for vector machines. In a critique of Parafrase, the first major project to build a vectorizing compiler, Ellis states:

> Parafrase attempts to fit the available parallelism to the target architecture. Because the architectures cannot use fine-grained, operation-level parallelism, ... the transformations operate at a coarse, all-or-nothing level, ignoring anything that cannot fit the mold.[2]

Ellis observes that the difficulty of parallelization is affected by the machine architecture. (Part of the original motivation for VLIW machines was that the uniform architecture would simplify compilation [FERN84].) For a vector machine, a compiler must produce a vector program, but relatively few of all possible parallelizations of a given program are vector programs. Thus, vectorizing transformations must be coarse. This has an adverse impact on applicability. In

[1]As observed in Chapter 6, trace scheduling requires enormous unrollings to approach the performance of perfect pipelining.

[2]From [Ell85], p. 21.

general, the coarser a transformation is, the harder the transformation is for a compiler to use. Coarse transformations are rigid—while very powerful, they can only be applied under restrictive conditions. This, in turn, results in a proliferation of transformations. Special transformations must be written to bring commonly occurring program constructs into a form suitable for manipulation by the vectorizing transformations. This adds to the complexity of the overall system, making it even harder to use effectively.

This line of reasoning paints a surprisingly accurate description of vectorizing compilers. For example, Parafrase consists of many coarse transformations loosely coordinated in global algorithms. The Parafrase documentation lists about twenty modules which implement one or more transformations each [Lea85]. While not all of these are intended for vectorization, a conservative count shows more than a dozen major and minor transformations that are useful for vectorization.[3] A good example of a transformation which is implemented in Parafrase but is unnecessary in a compaction-based system is loop rerolling. Loop rerolling heuristically searches for loops that have been unrolled by the programmer and rerolls them. This transformation might be worth implementing in a vectorizing compiler because loop unrolling is a standard optimization for sequential machines (and therefore many existing programs have unrolled loops) and because loop unrolling inhibits vectorization.

The highly structured parallelism exploited by vector machines limits the expressiveness of vectorization—some code does not vectorize. Estimates vary, but it is commonly claimed that 75% of all code vectorizes. This sounds impressive, but, by Amdahl's law, if 25% of a computation cannot be parallelized, then the potential speedup is a factor of four.

In contrast, compaction applies uniformly to the entire program. Because the tightly-coupled, synchronous machine model supports fine-grain parallelization,

[3]This does not include standard optimizations for sequential compilation.

a compaction-based compiler can be both simpler and more expressive than a vectorizing compiler.

13.3 The Future

The need for parallelization is often motivated by an economic argument: There are millions of lines of code written for sequential machines, so the best way to exploit new parallel machines is to write compilers that automatically restructure existing software. This reasoning is sound and the desire to reuse expensive software is sensible; however, this argument suggests that when Fortran is superseded by parallel programming languages, as it eventually will be, parallelization may also disappear.

We feel that the ideas presented here will be of lasting value. When high-level languages were introduced, the compiler took over the task of generating good code at the lowest level. Compiler optimizations, while not as good as a diligent and talented programmer, proved adequate for this task. The benefits of moving to high-level languages were great—creating a working program became much easier. We view compaction in the same light as traditional compiler optimizations. When programming languages suited to parallelism come into use, there will still be much parallelism that is too low-level or irregular to exploit by hand. Compaction techniques are ideally suited to exploiting this type of parallelism.

Appendix A

Miscellaneous Details

A.1 Computation of $\rho(r, w)$

We consider computing $\rho(r, w)$ for references of the form $A[aI + b]$, where I is an iteration variable and a and b are integers. In [WB87], Wolfe and Banerjee show that there is an exact dependency test for this case, and that all possible conflicting references can be enumerated. The algorithm in [WB87] provides all of the information needed to compute $\rho(r, w)$.

For illustration, we derive an algorithm to compute $\rho(r, w)$ for the most interesting cases—when $r \ ?_\delta \ w$ or $r \ ?{=}_\delta \ w$. Although reads and writes may occur in either order in these cases, it is still possible to have strict unrollings greater than one. As an example, assume the statement $A[10x] := A[x]$ appears in L. A graph of the time functions for the read and write references is shown in Figure A.1. Two conflicting references are shown. One is at element zero, but it occurs in the same iteration (time zero) for both references and is therefore preserved by any quantization. The other conflict is at element ten. Element ten is written by $A[10x]$ at time one and read by $A[x]$ at time ten. Thus unrolling the loop nine times preserve strictness. This follows because conflicting references in the positive direction must be more than nine iterations apart. A symmetric argument shows the same to be true in the negative direction.

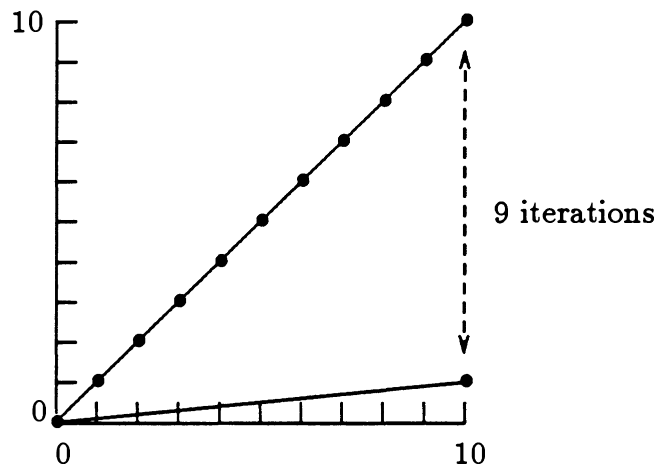


Figure A.1: Comparing references $A[10x]$ and $A[x]$.

Note that if the lower bound of x were something other than one the loop could be unrolled more and still preserve strictness. If, for instance, the loop began with $x = 0$ the greatest strictness-preserving unrolling would be ten. If the loop began at $x = 2$ the conflict at element ten would not exist and the unrolling could be as great as eighteen (the next conflict is at element twenty). The method we present can be easily extended to take advantage of knowledge of the upper and lower bounds of iteration variables; for the moment we continue to assume nothing is known about these bounds.

The generalization of the example requires a few simple facts about integer-valued linear functions. We state these facts without proof.

Definition A.1.1 Let $r(x) = ax + b$, where a and b are rationals. We define the set of *integer solutions* of r to be

$$\mathcal{I}(r) = \{y \mid y \in \mathcal{N} \land ay + b \in \mathcal{N}\}$$

There are linear expressions without integer solutions—e.g., $x + \frac{1}{2}$. We exclude these from consideration.

Fact A.1.1.1 Let $r(x) = ax + b$, where a and b are rationals and $a \neq 0$. Let x_1 be any integer solution of r. Let x_2 be the smallest integer solution of r greater than x_1. Then

$$\mathcal{I}(r) = \{a(x_1 + \Delta(x_2 - x_1)) + b | \Delta \in \mathcal{N}\}$$

We write $\hat{r}$ for the *period* $(x_2 - x_1)$ of r.

Fact A.1.1.2 Let $r(x) = a_r x + b_r$ and $w(x) = a_w x + b_w$ be two linear expressions.

$$\begin{array}{ll} \text{If} & \mathcal{I}(r) \cap \mathcal{I}(w) \neq \emptyset \\ \text{then} & \mathcal{I}(r) \cap \mathcal{I}(w) = \{\Delta \cdot lcm(\hat{r}, \hat{w}) + z | \Delta \in \mathcal{N}\} \\ \text{where} & z \in \mathcal{I}(r) \cap \mathcal{I}(w) \end{array}$$

We now describe the computation of the greatest strict unrolling when $r \mathrel{?_\delta} w$ (or $?{=}_\delta$). From the previous discussion, the objective is to compute the minimum vertical distance between r^{-1} and w^{-1} at a common integer solution; this corresponds to the "closest" conflicting reference. More precisely, if $r \mathrel{?_\delta} w$ or $r \mathrel{?{=}_\delta} w$ then $\min_{j \in \mathcal{I}(r) \cap \mathcal{I}(w)} |r^{-1}(j) - w^{-1}(j)|$ may be computed as follows:

1. Compute any integer solution k in $\mathcal{I}(r) \cap \mathcal{I}(w)$. This can be done using standard diophantine equation analysis.

2. Compute $\alpha = lcm(\hat{r}, \hat{w})$.

3. Compute p, the intersection point of r^{-1} and w^{-1}.

4. Using k and α, compute the smallest common integer solution greater than p. Call this g. The formula for g is:

   $$\begin{array}{ll} g = k + \alpha \left\lceil \frac{p-k}{\alpha} \right\rceil & \text{if } p \text{ is not a common integer solution} \\ g = p + \alpha & \text{if } p \text{ is a common integer solution} \end{array}$$

5. Compute the largest common integer solution less than p. This is

   $$\begin{array}{ll} h = g - \alpha & \text{if } p \text{ is not a common integer solution} \\ h = p - \alpha & \text{if } p \text{ is a common integer solution} \end{array}$$

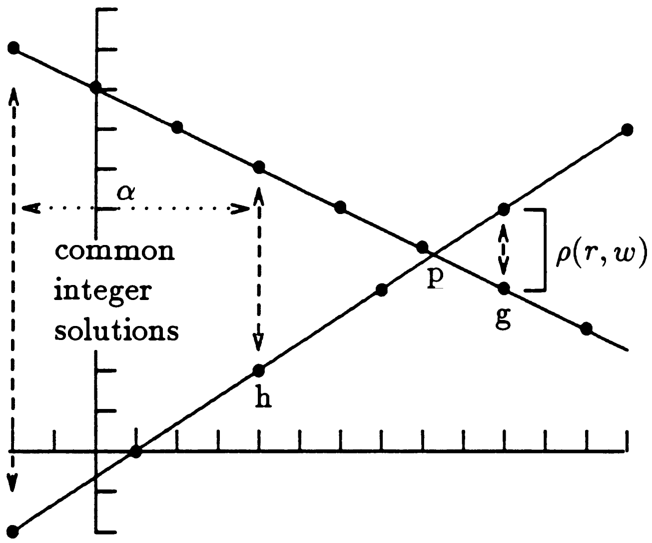


Figure A.2: Computation of $\rho(r, w)$.

6. Set $\rho(r, w) = \min(|r^{-1}(g) - w^{-1}(g)|, |r^{-1}(h) - w^{-1}(h)|)$.

Figure A.2 illustrates the values computed by the algorithm.

We do not present a proof of correctness for this algorithm. The idea of the proof is that the smallest distance in iterations must occur near the intersection point of the two lines. In fact, it must be with $lcm(\hat{r}, \hat{w})$ on either side of the intersection point. Thus, we need only consider the two common integer solutions lying on either side of the intersection.

A.2 Extensions to the Maximal Strict Quantization Algorithm

Unfortunately, $\rho(r, w)$ does not provide enough information to extend the algorithm for computing a maximal strict quantization to handle $?_\delta$.

Definition A.2.1 Let

$$\rho^<(r, w) = \max_{z \in \mathcal{Z}^+ \cup \{\infty\}} \forall x \in \mathcal{N}\ r^{-1}(x) < w^{-1}(x) \land Q \Rightarrow \left\lfloor \frac{r^{-1}(x)}{z} \right\rfloor < \left\lfloor \frac{w^{-1}(x)}{z} \right\rfloor$$

$$\rho^{>}(r,w) = \max_{z \in \mathcal{Z}^+ \cup \{\infty\}} \forall x \in \mathcal{N}\ r^{-1}(x) > w^{-1}(x) \wedge Q \Rightarrow \left\lfloor \frac{r^{-1}(x)}{z} \right\rfloor > \left\lfloor \frac{w^{-1}(x)}{z} \right\rfloor$$

where Q is the predicate $r^{-1}(z) \in \mathcal{N} \wedge w^{-1}(z) \in \mathcal{N}$.

The values $\rho^{<}(r,w)$ and $\rho^{>}(r,w)$ are the arguments of the minimum in step six of the algorithm to compute $\rho(r,w)$ when $r\ ?_\delta\ w$. We extend the definition of distance tables to include pairs $\langle \rho^{<}(r,w), \rho^{>}(r,w) \rangle$ for $?_\delta$ entries in a compatible comparison table. The following lemma extends Lemma 9.3.1 for $?_\delta$ and motivates the need for $\langle \rho^{<}, \rho^{>} \rangle$ pairs in the distance table.

Lemma A.2.2 Let c and d be compatible comparison and distance vectors with $c_i = ?_\delta$ or $?{=}_\delta$. Assume c_i is computed from r_i and w_i. Let q be a quantization of L. If $q_i > \rho^{<}(r_i, w_i)$ (resp. $\rho^{>}(r_i, w_i)$), then there is an x s.t. $x_i^c = <_o$ ($>_o$) in the original loop and $x_i^c = =_o$ in the quantized loop.

Proof: The lemma follows from the definition of $\rho^{<}(r_i, w_i)$ and $\rho^{>}(r_i, w_i)$. □

Consider the comparison and distance tables:

$$C = [?_\delta\ >_\delta\ ]\ D = [\langle 2,1 \rangle\ 2]$$

One strict quantization is obtained by unrolling strictly on the first dimension and arbitrarily on the second: $q_{max} = \langle 1\ \infty \rangle$. Another possibility is $q_{max} = \langle 2\ 2 \rangle$; the first component guarantees the strictness for locations x where $x_1^c = <_o$, the second component guarantees the strictness for all other locations. This is why more information is needed to handle $?_\delta$; the fact that $\langle 2\ 2 \rangle$ is a maximal strict quantization cannot be inferred if the distance entry for $?_\delta$ was $\rho(r,w)$.

Now consider the tables:

$$C = [?_\delta\ >_\delta\ >_\delta\ >_\delta]\ D = [\langle 2,1 \rangle\ 2\ 2\ 2]$$

A best quantization is $\langle 1\ \infty\ \infty\ \infty \rangle$. The quantization $\langle 2\ 2\ \infty\ \infty \rangle$ is maximal strict for this example, as are $\langle 2\ \infty\ 2\ \infty \rangle$ and $\langle 2\ \infty\ \infty\ 2 \rangle$. In this example, every position can act like a defining position. The $?_\delta$ component is the most important—it demands a strict unrolling that at least protects the read/write order for locations x where $x_1^c = <_o$. However, because all the other components in the table are $>_\delta$, the first dimension need not be unrolled to ensure strictness for locations x where $x_1^c = >_o$. This can be done by unrolling strictly on any other dimension. The following lemma shows that $\rho(r, w)$ does provide enough information for $?{=}_\delta$.

Lemma A.2.3 *Let $r \; ?{=}_\delta \; w$. Then $\rho^<(r, w) = \rho^>(r, w)$.*

Proof: A simple geometric argument based on the fact that the intersection point of r^{-1} and w^{-1} is a common integer solution. □

Consider the tables:

$$C = [?{=}_\delta \;\; ?{=}_\delta \;\; >_\delta \; >_\delta] \, D = [1 \; 1 \; 1 \; 1]$$

There are only two maximal quantizations for this example: $q_{max} = \langle 1 \; 1 \; 1 \; \infty \rangle$ or $q_{max} = \langle 1 \; 1 \; \infty \; 1 \rangle$.

This deserves some explanation. If q_1 were greater than one, then, by Lemma A.2.2, there would be a location x where $x_1^c = >_o$ in the original loop and $x_1^c = =_o$ in the quantized loop. Therefore we may choose values of the iteration variables of the quantized loop such that $x^c = \langle =_o \; <_o \; \ldots \rangle$; an unrolling greater than one for q_1 interchanges the read/write order for at least one point. A similar argument shows that q_2 must be one.

One of q_3 and q_4 must be one. This follows because the first two components of the comparison vector are $?{=}_\delta$. By definition, there is a location x such that $x^c = \langle =_o \; =_o \; \ldots \rangle$. Guaranteeing strictness here requires a strict unrolling in either the third or fourth component.

Most of the lemmas and definitions presented in Chapter 9 require only trivial modifications to allow for $?_\delta$ and $?{=}_\delta$. One that requires considerable change is the definition of a defining position.

Definition A.2.4 Let c be a comparison vector of length n. Let j be the least index such that c_j is not $=_\delta$ or $?{=}_\delta$.

- If there is no such j, then c's only defining position is $n+1$.
- If $c_j \neq ?_\delta$, then c_k is a defining position for c if $c_k = c_j \wedge \forall z, j < z < k, c_z \in \{=_\delta, c_j\}$
- If $c_j = ?_\delta$, then j is the *primary* defining position of c. Let c_i be the least index greater than j that is not $=_\delta$ or $?{=}_\delta$. If c_i is either $<_\delta$ or $>_\delta$, then c_k is a *secondary* defining position of c if $c_k = c_i \wedge \forall z, i < z < k, c_z \in \{=_\delta, c_i\}$

The following lemmas may be proved by straightforward application of Lemmas 9.3.1 and A.2.2.

Lemma A.2.5 Let c and d be compatible comparison and distance vectors. Let j be the least defining position of c. If there is a position c_k s.t. $k < j \wedge c_k = ?{=}_\delta$, then any strict quantization q must satisfy $q_k \leq d_k$.

Lemma A.2.6 Let c and d be compatible comparison and distance vectors. Let j be the least defining position of c and let $c_j = ?_\delta$. Let $d_j = \langle \rho^<, \rho^> \rangle$. Then any strict quantization q must satisfy one of the following:

1. $q_j \leq \min(\rho^<, \rho^>)$
2. $q_j \leq \rho^<$ and there is a secondary defining position c_k s.t. $c_k = >_\delta$ and $q_k \leq d_k$.
3. $q_j \leq \rho^>$ and there is a secondary defining position c_k s.t. $c_k = <_\delta$ and $q_k \leq d_k$.

The revised algorithm is shown in Figure A.3. The proof of correctness is similar to the proof for the previous algorithm. Lemmas A.2.5 and A.2.6 are used to cover the new cases.

```
Input: C   m × n comparison table
       D   m × n distance table

declare array defined[1..m], unroll[1..n], q[1..n]
∀i, j if D[i, j] = ⟨ρ<, ρ>⟩
    then select(D[i, j]) := min(ρ<, ρ>)
    else select(D[i, j]) := D[i, j];
∀i defined[i] := nil;
(* Pass 1 *)
for j = 1 to n do
    begin
        if ∀i(defined[i] ∨ C[i, j] = =δ)
            then q[j] := ∞
            else q[j] := min_{i|defined[i]=nil} select(D[i, j]);
        for i = 1 to n do if C[i, j] ∈ {<δ, >δ, ?δ} ∧ defined[i] = nil
                then defined[i] := j
    end;
(* Pass 2 *)
for j = n to 1 do
    for each vector C_i
        begin
            if j is a defining position of C_i, D[i, j] ≥ q[j], defined[i] < j,
                    and C[i, j] ≠ ?δ then
                begin
                    x := defined[i];
                    defined[i] := j;
                    q[x] := min_{i|defined[i]=x} D[i, x]
                end;
            if j is a defining position of C_i and C[i, j] = ?δ, and k is a
                    secondary defining position s.t. select(D[i, k]) ≤ q[k] then
                begin
                    let ⟨ρ<, ρ>⟩ = D[i, j];
                    if C[i, k] = >δ then select(D[i, j]) = ρ<;
                    if C[i, k] = <δ then select(D[i, j]) = ρ>;
                    q[j] := min_{i|defined[i]=j} select(D[i, j])
                end
        end;
```

Figure A.3: Extended algorithm for computing a maximal strict quantization.

A.3 Proof of Correctness of Mitred Quantization

Lemma A.3.1 Let L be two nested loops, and let $\hat{L}$ be a mitred quantization of L. Then L sim $\hat{L}$.

Proof: We must show that dependent iterations are executed in the same order in L and $\hat{L}$. Let $\langle a, b\rangle$ and $\langle a+c_a, b+c_b\rangle$ be dependent iterations in L. We show that

$$\langle a+c_a, b+c_b\rangle < \langle a, b\rangle \Rightarrow$$
$$\left\langle \left\lfloor \frac{a+c_a}{\Delta i+1} \right\rfloor, \left\lfloor \frac{b+c_b+\left\lfloor \frac{(a+c_a)\bmod(\Delta i+1)}{-s} \right\rfloor}{\Delta j+1} \right\rfloor \right\rangle \le \left\langle \left\lfloor \frac{a}{\Delta i+1} \right\rfloor, \left\lfloor \frac{b+\left\lfloor \frac{a\bmod(\Delta i+1)}{-s} \right\rfloor}{\Delta j+1} \right\rfloor \right\rangle$$

Clearly, $c_a \le 0$, as $a \ge a + c_a$. Therefore, $\left\lfloor \frac{a+c_a}{\Delta i+1} \right\rfloor \le \left\lfloor \frac{a}{\Delta i+1} \right\rfloor$. If $c_b \le 0$ the lemma holds, so assume that $c_b > 0$. Because s is the dependency with minimum negative slope, it follow that $s = \frac{c_a}{kc_b}$ where $k \ge 1$. Now, if $\left\lfloor \frac{a+c_a}{\Delta i+1} \right\rfloor < \left\lfloor \frac{a}{\Delta i+1} \right\rfloor$ the lemma holds, so assume that the two expressions are equal. This implies that $(a+c_a) \bmod (\Delta i+1) \le a \bmod (\Delta i+1)$. We use this fact to rewrite the second component of the iteration vectors. Let $p \equiv a \bmod (\Delta i+1)$. We must show that

$$\left\lfloor \frac{b+c_b+\left\lfloor \frac{p}{-s} \right\rfloor}{\Delta j+1} \right\rfloor \le \left\lfloor \frac{b+\left\lfloor \frac{p-c_a}{-s} \right\rfloor}{\Delta j+1} \right\rfloor$$

Now,

$$\frac{p-c_a}{-s} = \frac{(p-c_a)\,c_b}{-c_a} = \frac{kpc_b - kc_ac_b}{-c_a} = \frac{kpc_b}{-c_a} + kc_b$$

and similarly

$$\frac{p}{-s} = \frac{kpc_b}{-c_a}$$

So the previous inequality becomes

$$\left\lfloor \frac{b+c_b+\left\lfloor \frac{kpc_b}{-c_a} \right\rfloor}{\Delta j+1} \right\rfloor \le \left\lfloor \frac{b+kc_b+\left\lfloor \frac{kpc_b}{-c_a} \right\rfloor}{\Delta j+1} \right\rfloor$$

which holds because $k \ge 1$ and $c_b \ge 0$. □

A.4 Conservative Dependency Analyzers Form a Complete Lattice

Theorem A.4.1 The domain of conservative dependency analyzers is a complete lattice.

Proof: To prove that a domain is a complete lattice, it is sufficient to show that every subset of the domain has a least upper bound in the domain [Sto77]. Consider any set $\mathcal{D}$ of conservative dependency analyzers. Let D be the dependency analyzer where $D(x, y, s) = \top$ if and only if for some $D' \in \mathcal{D}$, $D'(x, y, s) = \top$. Clearly, for all $D' \in \mathcal{D}$, $D' \leq D$, so this is an upper bound for the set. Also, $D(x, y, s) = \top \Rightarrow D^{\mathrm{perf}}(x, y, s) = \top$, so D is conservative.

Let D' be any other upper bound for $\mathcal{D}$. We claim that $D \leq D'$. Assume $D'(x, y, s) = \bot$. Then there is no $D'' \in \mathcal{D}$ such that $D''(x, y, s) = \top$. Therefore, $D(x, y, s) = \bot$, so $D \leq D'$. □

Bibliography

[AAG+86] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, K. Sarocky, and J. A. Webb. Warp architecture and implementation. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 346–356, June 1986.

[ABC+87] F. E. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. In *Proceedings of the 1987 International Conference on Supercomputing*, 1987.

[AC72] F. E. Allen and J. Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1972.

[AC76] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–147, March 1976.

[AHU74] A. V. Aho, J. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AI83] Arvind and R. A. Iannucci. A critique of multiprocessing von Neumann style. In *Proceedings of the 10th International Symposium on Computer Architecture*, pages 426–436, Stockholm, Sweden, June 1983.

[AK82] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. In Kai Hwang, editor, *Supercomputers: Design and Applications*, pages 188–203. IEEE Computer Society Press, Silver Spring, MD, 1982.

[AK84] J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, volume 19, pages 233–246, June 1984.

[AKPW83] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 1983 Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[All74] F. E. Allen. Interprocedural data flow analysis. In *Proceedings of the 1974 IFIP Congress*, pages 398–402, 1974.

[All75] F. E. Allen. Bibliography on program optimization. Technical Report RC-5767, IBM T. J. Watson Research Center, 1975.

[All83] J. R. Allen. *Dependence Analysis for Subscripted Variables and its Application to Program Transformations.* Ph.D. dissertation, Rice University, 1983.

[All85] Alliant Computer Systems Corporation, Acton Mass. *Product Summary*, January 1985.

[Amd67] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS 1967 Spring Joint Computing Conference*, volume 30, pages 483–487, 1967.

[AN87a] A. Aiken and A. Nicolau. Loop Quantization: An analysis and algorithm. Technical Report 87-821, Cornell University, March 1987.

[AN87b] A. Aiken and A. Nicolau. Perfect Pipelining: A new loop parallelization technique. Technical Report 87-873, Cornell University, October 1987.

[AN88a] A. Aiken and A. Nicolau. A development environment for horizontal microcode. *IEEE Transactions on Software Engineering*, 14(5):584–594, May 1988.

[AN88b] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1988.

[AN88c] A. Aiken and A. Nicolau. Perfect Pipelining: A new loop parallelization technique. In *Proceedings of the 1988 European Symposium on Programming*, pages 221–235. Springer Verlag Lecture Notes in Computer Science no. 300, March 1988. Also available as Cornell Technical Report TR 87-873.

[ASKL81] W. Abu-Sufa, D. J. Kuck, and D. H. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Transactions on Computers*, C-30(5):341–356, May 1981.

[AST67] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The ibm system/360 model 91: Machine philosophy and instruction handling. *IBM Journal of Research and Development*, pages 8–24, January 1967.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[Bae73] J. L. Baer. A survey of some theoretical aspects of multiprocessing. *ACM Computing Surveys*, 5(1):31–80, March 1973.

[Ban76] U. Banerjee. Data dependence in ordinary programs. Masters dissertation, University of Illinois at Urbana-Champaign, November 1976. Dept. of Computer Science Rpt. 76-837.

[Ban79] U. Banerjee. *Speedup of Ordinary Programs.* Ph.D. dissertation, University of Illinois at Urbana-Champaign, October 1979. Dept. of Computer Science Rpt. 79-989.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, 1984.

[BBK+68] G. Barnes, R. Brown, M. Kato, D. J. Kuck, D. Slotnick, and R. Stokes. The ILLIAC IV computer. *IEEE Transactions on Computers*, C-17(8):746–757, August 1968.

[BC86] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, volume 21, pages 162–175, July 1986.

[BCKT79] U. Banerjee, S. Chen, D. J. Kuck, and R. Towle. Time and parallel processor bounds for FORTRAN-like loops. *IEEE Transactions on Computers*, C-31(1):78–81, January 1979.

[Ber66] A. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, October 1966.

[BF67] H. W. Bingham and D. A. Fisher. Automatic detection of parallelism in computer programs. Technical Report TR-67-4, AD 622 274, Burroughs, November 1967. Cited in [Bae73].

[BH87] P. Brinch Hansen. Joyce—a programming language for distributed systems. *Software—Practice and Experience*, 17(1):29–50, January 1987.

[Cal87] D. Callahan. *A Global Approach to Detection of Parallelism.* Ph.D. dissertation, Rice University, 1987.

[CCD+87] P. Clancy, B. F. Cutler, J. C. Dodd, D. W. Gilmore, R. P. Nix, J. J. O'Donnell, and C. P. Ryland. UNIX[1] on a VLIW. In *Proceedings of the USENIX Technical Conference and Exhibition*, Phoenix, Arizona, June 1987.

[CCK87] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined architectures. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 295–304, August 1987.

[CF87] R. Cytron and J. Ferrante. What's in a name: The value of renaming for parallelism detection and storage allocation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, August 1987.

[Cha81] A. E. Charlesworth. An approach to scientific array processing: The architectural design of the AP-120b/FPS-164 family. *IEEE Computer*, 14(3):18–27, 1981.

[CKV85] R. Cytron, D. J. Kuck, and A. V. Veidenbaum. The effect of restructuring compilers on program performance for high-speed computers. *Computer Physics Communications*, 37:37–48, 1985.

[CNO+87] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987.

[CP86] D. Padua C. Polychronopoulos, D. J. Kuck. Execution of parallel loops on parallel processor systems. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 519–527, August 1986.

[Cyd87] Cydrome Inc., Palo Alto, Ca. *Technical Summary*, 1987.

[1]UNIX is a registered trademark of AT&T Technologies.

[Cyt84] R. Cytron. *Compile-time Scheduling and Optimization for Asynchronous Machines.* Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1984.

[Cyt86] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836–844, August 1986.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[DT76] S. Dasgupta and J. Tartar. The identification of maximal parallelism in straight-line microprograms. *IEEE Transactions on Computers*, C-25:986–991, October 1976.

[Ebc87] K. Ebcioğlu. A compilation technique for software pipelining of loops with conditional jumps. In *Proceedings of the 20th Annual Workshop on Microprogramming*, pages 69–79, December 1987.

[Ell85] J. R. Ellis. *Bulldog—A Compiler for VLIW Architectures.* Ph.D. dissertation, Yale University, February 1985.

[FERN84] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: A smart compiler and a dumb machine. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, pages 37–47, June 1984.

[Fis79] J. A. Fisher. *The Optimization of Horizontal Microcode within and beyond Basic Blocks: an Application of Processor Scheduling with Resources.* Ph.D. dissertation, New York University, 1979.

[Fis80] J. Fisher. 2^n-way jump microinstruction hardware and an effective instruction binding method. In *Proceedings of the 13th Annual Workshop on Microprogramming*, pages 64–75, December 1980.

[Fis81] J. A. Fisher. Trace Scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–90, July 1981.

[Fis83] J. A. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the Tenth Annual Symposium on Computer Architecture*, pages 140–150, Stockholm, June 1983.

[FLS81] J. Fisher, D. Landskov, and B. Shriver. Microcode compaction: Looking backward and looking forward. In *Proceedings of the 1981 AFIPS National Computer Conference*, volume 50, pages 95–102, 1981.

[Fly66] M. J. Flynn. Very high speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[FM85] J. Fortes and D. Moldovan. Parallelism detection and transformation techniques useful for VLSI algorithms. *Journal of Parallel and Distributed Computing*, 2(3):277–301, August 1985.

[FO84] J. A. Fisher and J. J. O'Donnell. VLIW machines: Multiprocessors we can actually program. In *Proceedings of CompCon Spring 84*, pages 299–305. IEEE Computer Society, February 1984.

[For86] American National Standards Institute, American National Standards Institutem, New York. *American National Standard for Information Systems. Programming Language Fortran.S8 (X3.9-198x). Revision of X3.9-1978. Draft S8, Version 99.*, 1986.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, June 1987.

[Gil58] S. Gill. Parallel programming. *Computer Journal*, 1(1):2–10, April 1958.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[GS87] R. Gupta and M. Soffa. Detecting and redistributing parallelism in program regions. Technical report, University of Pittsburgh, 1987.

[Gue79] I. Guessarian. Program transformations and algebraic semantics. *Theoretical Computer Science*, 9(9):39–65, 1979.

[HK84] R. T. Hood and K. Kennedy. A programming environment for Fortran. Technical Report COMP TR 84-1, Rice University, 1984.

[HL82] R. W. Heuft and W. D. Little. Improved time and parallel processor bounds for Fortran-like loops. *IEEE Transactions on Computers*, C-31(1), January 1982.

[HPR88a] S. Horowitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. In *Proceedings of the 1988 SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 133–145, January 1988.

[HPR88b] S. Horowitz, J. Prins, and T. Reps. On the adequacy of dependence graphs for representing programs. In *Proceedings of the 1988 SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 146–157, January 1988.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[Hym66] H. Hyman. Comments on a problem in concurrent programming control. *Communications of the ACM*, 9(1):45, January 1966.

[Jou87] P. Jouvelot. Semantic parallelization: A practical exercise in abstract interpretation. In *Proceedings of the 1987 SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 39–48, January 1987.

[Ken87] K. Kennedy. Compiling scientific programs for execution on parallel machines. Notes from a lecture presented at the 1987 SIGACT-SIGPLAN Symposium on Principles of Programming Languages, January 1987.

[KKLW80] D. J. Kuck, R. Kuhn, B. Leasure, and M. Wolfe. The structure of an advanced vectorizer for pipelined processors. In *Proceedings of the 4th Int'l Computer Software and Applications Conference*, pages 709–715, October 1980.

[KKP+81] D. J. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 1981 SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[KM75] D. J. Kuck and Y. Maruyama. Time bounds on the parallel evaluation of arithmetic expressions. *SIAM Journal of Computing*, 4:147–62, June 1975.

[KMC72] D. J. Kuck, Y. Muraoka, and S. C. Chen. On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up. *IEEE Transactions on Computers*, C-21(12):1293–1310, December 1972.

[KN85] K. Karplus and A. Nicolau. Efficient hardware for multi-way jumps and pre-fetches. In *Proceedings of the 18th Annual Workshop on Microprogramming*, pages 11–18, December 1985.

[Knu66] D. E. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5):321–322, May 1966.

[Kog77] P. M. Kogge. The microprogramming of pipelined processors. In *Proceedings of the 4th Annual International Symposium on Computer Architecture*, 1977.

[Kuc68] D. J. Kuck. ILLIAC IV software and application programming. *IEEE Transactions on Computers*, C-17(8):758–770, August 1968.

[Kuc76] D. J. Kuck. Parallel processing of ordinary programs. In *Advances in Computers*, volume 15, pages 119–179. Academic Press, New York, 1976.

[Kuc78] D. J. Kuck. *The Structure of Computers and Computations*, volume 1. John Wiley and Sons, New York, 1978.

[Kuh80] R. H. Kuhn. *Optimization and Interconnection Complexity for: Parallel Processors, Single-Stage Networks, and Decision Trees.* Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1980. Dept. of Computer Science Rpt. 80-1009.

[LA82] J. Lah and D. Atkins. Tree compaction of microprograms. In *Proceedings of the 16th Annual Workshop on Microprogramming*, pages 23–33, October 1982.

[Lam74] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[Lam87] M. Lam. *A Systolic Array Optimizing Compiler.* Ph.D. dissertation, Carnegie Mellon University, 1987.

[LDS80] D. Landskov, S. Davidson, and B. Shriver. Local microcode compaction techniques. *ACM Computing Surveys*, 12(3):261–294, September 1980.

[Lea85] B. Leasure. The Parafrase project's Fortran analyzer major module documentation. Technical Report CSRD No. 504, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, July 1985.

[Mal78] P. Mallett. *Methods of Compacting Microprograms.* Ph.D. dissertation, University of Southwestern Louisiana, Lafayette, 1978.

[McM83] F. H. McMahon. Lawrence Livermore National Laboratory FORTRAN kernels: MFLOPS. Livermore, CA., 1983.

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, pages 96–103, 1979.

[MS87] A. Munshi and B. Simons. Scheduling sequential loops on parallel processors. Technical Report 5546, IBM, 1987.

[Mul87] Multiflow Computer Inc., Branford, Connecticut. *Technical Summary*, 1987.

[Mur71] Y. Muraoka. *Parallelism Exposure and Exploitation in Programs.* Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1971.

[NF84] A. Nicolau and J. Fisher. Measuring the parallelism available for Very Long Instruction Word architectures. *IEEE Transactions on Computers*, C-33:968–76, November 1984.

[Nic84] A. Nicolau. *Parallelism, Memory Anti-Aliasing, and Correctness for Trace Scheduling Compilers.* Ph.D. dissertation, Yale University, 1984.

[Nic85a] A. Nicolau. Loop Quantization, or unwinding done right. Technical Report 85-709, Cornell University, 1985.

[Nic85b] A. Nicolau. Uniform parallelism exploitation in ordinary programs. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 614–618, August 1985.

[Nic88] A. Nicolau. Runtime disambiguation: Coping with statically unpredictable references. *IEEE Transactions on Computers*, C-37(12), December 1988. To Appear.

[Nie85] F. Nielson. Program transformations in a denotational setting. *ACM Transactions on Programming Languages and Systems*, 7(3):359–379, July 1985.

[NPA88] A. Nicolau, K. Pingali, and A. Aiken. Fine-grain compilation for pipelined machines. *Journal of Supercomputing*, August 1988. To appear.

[NPD87] A. Neirynck, P. Panangaden, and A. J. Demers. Computation of aliases and support sets. In *Proceedings of the 1987 SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 274–283, January 1987.

[Num83] Numerix Corporation, Newton, Massachussetts. *MARS-432 Programmer's Reference Manual*, 1983. Cited in [Ell85].

[OO[illegible]] K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT-SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, April 1984.

[PA86] K. Pingali and Arvind. Efficient demand-driven evaluation. Part 2. *ACM Transactions on Programming Languages and Systems*, 8(1):109–139, January 1986. Part 1 appears in the April, 1985 issue.

[Pai83] R. Paige. Transformational programming—applications to algorithms and systems. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 73–87, January 1983.

[Pat85] D. A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, January 1985.

[PH79] D. A. Padua-Haiek. *Multiprocessors: Discussion of Some Theoretical and Practical Problems.* Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1979.

[PK82] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[Plo] G. D. Plotkin. A structural approach to operational semantics. Text prepared at University of Aarhus.

[Pol87] C. Polychronopulos. Advanced loop optimizations for parallel computers. Technical Report CSRD No. 664, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, June 1987.

[Ree84] A. P. Reeves. Parallel pascal: An extended pascal for parallel computers. *Journal of Parallel and Distributed Computing*, 1, 1984.

[RF72] E. Riseman and C. Foster. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computers*, C-21(12):1405–1411, December 1972.

[RG69] C. Ramamoorthy and M. Gonzalez. A survey of techniques for recognizing parallel processable streams in computer programs. In *Proceedings of the AFIPS 1969 Fall Joint Computer Conference*, pages 1–15, 1969.

[RG81] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Workshop on Microprogramming*, pages 183–198, October 1981.

[Rog87] H. Rogers Jr. *Theory of Recursive Functions and Effective Computability.* MIT Press, 1987.

[RT74] C. Ramamoorthy and M. Tsuchiya. A high-level language for horizontal microprogramming. *IEEE Transactions on Computers*, C-23:791–801, August 1974.

[RTT73] C. Ramamoorthy, M. Tabendeh, and M. Tsuchiya. A higher level language for micro-programming. In *Proceedings of the 6th Annual Workshop on Microprogramming*, pages 139–144, September 1973.

[Rus69] E. C. Russell. *Automatic Program Analysis.* Ph.D. dissertation, University of California, Los Angeles, 1969. Cited in [Bae73].

[Rut88] J. C. Ruttenberg. Private communication., February 1988.

[SBI+70] B. T. Smith, J. M. Boyle, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines: EISPACK Guide.* Springer-Verlag, New York, 2 edition, 1970.

[SDJ84] B. Su, S. Ding, and L. Jin. An improvement of trace scheduling for global microcode compaction. In *Proceedings of the 17th Annual Workshop on Microprogramming*, pages 78–85, December 1984.

[SDX86] B. Su, S. Ding, and J. Xia. Urpr—an extension of urcr for software pipelining. In *Proceedings of the 19th Annual Workshop on Microprogramming*, pages 104–108, October 1986.

[SDX87] B. Su, S. Ding, and J. Xia. GURPR—a method for global software pipelining. In *Proceedings of the 20th Annual Workshop on Microprogramming*, pages 88–96, December 1987.

[Shu77] L. J. Shustek. *Analysis and Performance of Computer Instruction Sets.* Ph.D. dissertation, Stanford, 1977. Cited in [Lam87].

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, volume 1 of *MIT Press Series in Computer Science.* MIT Press, 1977.

[TF70] G. S. Tjaden and M. J. Flynn. Detection and parallel execution of independent statements. *IEEE Transactions on Computers*, 19(10):889–895, October 1970.

[TG76] M. Tsuchiya and M. Gonzales. Toward optimization of horizontal microcode. *IEEE Transactions on Computers*, C-25:992–999, October 1976.

[Tou84] R. F. Touzeau. A Fortran compiler for the FPS-164 scientific computer. In *Proceedings of the 1984 ACM SIGPLAN Symposium on Compiler Construction*, pages 48–57, June 1984.

[Tow76] R. A. Towle. *Control and Data Dependence for Program Transformations.* Ph.D. dissertation, University of Illinois at Urbana-Champaign, March 1976.

[TTT81] M. Tokoro, E. Tamura, and T. Takizuka. Optimization of microprograms. *IEEE Transactions on Computers*, C-30(7):491–504, July 1981.

[TTTY77] M. Tokoro, E. Tamura, T. Takizuka, and I. Yamaura. A technique of global optimization of microprograms. In *Proceedings of the 11th Annual Workshop on Microprogramming*, pages 41–50, 1977.

[WB87] M. Wolfe and U. Banerjee. Data dependence for parallelism detection. Technical report, Kuck and Associates, Inc., June 1987.

[Wei84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[Wol78] M. J. Wolfe. Techniques for improving the inherent parallelism in programs. Masters dissertation, University of Illinois at Urbana-Champaign, July 1978. Dept. of Computer Science Rpt. 78-929.

[Wol82] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Ph.D. dissertation, University of Illinois at Urbana-Champaign, October 1982.

[Wol86] M. Wolfe. Advanced loop interchanging. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 536–543, August 1986.

[Wol87] M. Wolfe. Loop skewing: the wavefront method revisited. Technical report, University of Illinois at Urbana-Champaign, April 1987.

[Woo78] G. Wood. On the packing of micro-opertions into micro-instruction words. In *Proceedings of the 11th Annual Workshop on Microprogramming*, pages 51–55, 1978.