Extracting Parallelism from Sequential Programs

by

Wilson Cheng-Yi Hsieh

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 1988

© Wilson C. Hsieh, 1988

The author hereby grants to MIT permission to reproduce and
to distribute copies of this thesis document in whole or in part.

Signature of Author ______________________________
Department of Electrical Engineering and Computer Science
6 May 1988

Certified by ______________________________
Michael Burke
Company Supervisor
IBM T.J. Watson Research Center

Certified by ______________________________
William E. Weihl
Assistant Professor of Electrical Engineering
Thesis Supervisor

Accepted by ______________________________
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Extracting Parallelism from Sequential Programs

by

Wilson Cheng-Yi Hsieh

Submitted to the Department of Electrical Engineering and Computer Science
on 6 May 1988, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Science in Electrical Engineering and Computer Science

Abstract

Current vectorizers and parallelizers all concentrate on transforming innermost `do` loops into vector or parallel loops; we present techniques for parallelizing nested `do` loops as well as blocks of code that are more general than the iterations of `do` loops. These techniques are based on a new representation for sequential programs, the *constrained forward control dependence graph*.

We generate nested parallelism in the form of parallel loops (`doall`) and parallel blocks (`cobegin/coend`). Our code generation incorporates an optimization, *privatization*, that improves the parallelism that we can achieve. Working off of the constrained forward control dependence graph, we present algorithms to insert some explicit synchronization, perform privatization, and generate parallel code.

Thesis Supervisor: William E. Weihl
Title: Assistant Professor of Electrical Engineering

Biography

The author is a member of Tau Beta Pi, Phi Beta Kappa, and Sigma Xi; he will be receiving his bachelor's and master's degrees this May from the Massachusetts Institute of Technology. He is a co-author of two papers written while he was on an internship assignment at IBM Research: "On the Automatic Generation of Useful Parallelism: A Tool and An Experiment (Extended Abstract)", with Michael Burke, Ron Cytron, Jeanne Ferrante, Dave Shields, and Vivek Sarkar, which was accepted for the 1988 ACM Sigplan Symposium on Parallel Programming: Experience with Applications, Languages, and Systems; and "Automatic Determination of Private and Shared Variables for Nested Processes", with Michael Burke, Ron Cytron, and Jeanne Ferrante, an IBM Technical Report. He will begin work toward his PhD at the Massachusetts Institute of Technology in the fall of 1988, with support from a National Science Foundation Graduate Fellowship.

Contents

Acknowledgements **8**

1 Introduction **10**

1.1 Input language 11

1.2 Output language 12

1.2.1 Private variables 12

1.2.2 Semaphores 14

1.3 Summary 14

2 Program analyses **18**

2.1 Control flow 18

2.2 Control dependence 21

2.3 Data dependence 26

2.3.1 Data flow analysis 29

3 Constrained forward control dependence **31**

3.1 Forward control dependence 32

3.1.1 Discussion 32

3.2 Data dependence constraints 36

3.2.1 Constraint edges 36

3.2.2 Constraint marks 38

3.2.3 Discussion 38

4 Algorithms **42**
4.1 Privatization 42
4.1.1 Privatizing blocks 45
4.1.2 Privatizing loops 52
4.2 Explicit synchronization 53
4.3 Code generation 59

5 Conclusions **62**

A Computing constraints **64**

List of Figures

1-1 Output language example 13
1-2 Semaphore example 15
1-3 Parallelization example 17

2-1 Sample program 19
2-2 Control flow graph of program in Figure 2-1 20
2-3 Graph grammar for structured programs 22
2-4 Control dependence graph of program in Figure 2-1 24
2-5 Data dependence example 28
2-6 Loop-independent and loop-carried dependences 28

3-1 Forward control flow graph of program in Figure 2-1 33
3-2 Forward control dependence graph of program in Figure 2-1 34
3-3 Constraint edge discussion 39
3-4 Forward control dependence graph of program in Figure 3-3(a) 40

4-1 Privatization example 43
4-2 Privatization versus renaming 44
4-3 No privatization due to flow dependences 45
4-4 Forward control dependence graph during block privatization 46
4-5 No privatization 47
4-6 Privatization examples 48
4-7 Privatization for arrays 50

4-8 No privatization 53
4-9 Synchronization example 54
4-10 Private semaphores 55
4-11 Non-removable constraint edge 56
4-12 Adding synchronization to the forward control dependence graph 57

Acknowledgements

Special thanks to Ron Cytron, who is responsible for many of the ideas in this thesis, and was always available to discuss things. Thanks also to the rest of the PTRAN group at IBM's T.J. Watson Research Center - Fran Allen, Michael Burke, Philippe Charles, Jeanne Ferrante, Vivek Sarkar, and Dave Shields - who were a wonderful group of people to work with. Thanks also to Cathy McCarthy, Francois Irigoin, and Jan Stone for being very friendly and helpful during my stay at IBM; thanks to Kip Fern, who was a good friend and roommate.

Thanks to Bill Weihl, who was a great thesis and graduate advisor; Albert Meyer, who was very helpful as my undergraduate advisor; Brian Oki, a great TA in 6.035 and good friend.

Thanks also to all those who read my thesis and provided comments: Bill Weihl, Ron Cytron, Michael Burke, Bob Gruber, and Jeff Cohen.

Thanks to Marcus Thompson, and all of the other musicians with whom I have played; without them, I would not have enjoyed MIT very much: Joyce Wong, Chung-Pei Ma, Richard Gotlib, Jee-Hoon Yap, Phil Hsu, Una Hwang, Barbara Hughey, Ellen Lin, Ken Goodson, Nina Chen, Monty McGovern, Marc Ryser, Sam Osofsky, Kirk Chao, Bob Hall, Bob Davis, Jean Rife, and many others.

Thanks also to my good friends at Burton House, who helped me survive this place: David Krakauer (who put up with me as lab partner for three terms), Louis Pepe and Jeff Finer (who helped turn the first triple play in Burton history), Kris Sheahan, Joanne Chee, Orlee Israeli, Sam Osofsky, Anne Okamura, Julie Tsai, Chris Adams, Chris Joerg,

Harold Stern, Mike Niles, and lots of others.

Thanks to Oliver Liang, Steve Liu, Dave Shen, and Terry Huang, who have been great friends since high school.

And, last but certainly not least, thanks and love to my parents, who have unswervingly helped me stumble through life; thank you for all of the love that you have given me.

Chapter 1

Introduction

As computers have steadily become smaller and faster, they have been used to solve ever larger problems. However, device sizes and speeds are already approaching physical limits, and there are still many problems that today's fastest computers cannot begin to solve in reasonable amounts of time. The obvious way to continue increasing the speed of computers is to "divide and conquer" problems with multiple CPUs. Many commercially available computers already have multi-CPU architectures, such as IBM's 3090, Alliant's FX-8, and Thinking Machines' Connection Machine. To make full use of the extra processing power in these machines, we need to develop parallel software.

Several research groups are building vectorizing or parallelizing compilers, which convert sequential language programs into vector or parallel language programs. This thesis deals with work on the PTRAN parallelizing system at IBM's T.J. Watson Research Center [ABCC87]; related projects include Parafrase I and II at the University of Illinois [KKLW80] and PFC at Rice University [AK87].

Several important reasons for working on such systems exist. A large base of software already exists, and it would be infeasible to discard it and write new software; we must be able to convert sequential software to run on parallel machines. Experience in building and using parallelizers should also be useful in developing solutions to several problems in parallel programming. Parallelizers may help in developing parallel program

debuggers or verifiers; debugging non-deterministic parallel programs is still an incredibly difficult task — perhaps the most serious problem in using parallel languages. Future parallel programming languages should ideally be portable, yet efficient on different parallel architectures; compilers will have to perform program restructuring to make this possible.

PTRAN, Parafrase, and PFC are "source-to-source" compilers: they translate high-level sequential languages into high-level parallel languages. PTRAN currently converts VS FORTRAN programs into Parallel Fortran programs [IBM88]. This approach has a major advantage over compiling to machine code: a PTRAN user can "hand-tune" the output program that PTRAN produces, if he possesses some knowledge about the input program that PTRAN cannot deduce.

Our work in the PTRAN system allows us to generate more parallelism than other systems; current vectorizers and parallelizers all concentrate on transforming innermost `do` loops into vector or parallel loops. Our program representation allows us to parallelize nested `do` loops, as well as blocks of code that are more general than the iterations of `do` loops. We generate nested parallelism; each parallel process can itself contain parallel processes.

Although the PTRAN system currently transforms only VS FORTRAN programs, our techniques for parallelization can be applied to most imperative programming languages. For the sake of exposition, we shall assume that we have abstract input and output languages as described in the following two sections.

1.1 Input language

Our input language (a sequential language) has only a single looping construct, `do` loops (similar to Fortran `do` loops). We use an `if-then-else` statement for conditional execution; multiway branches would not add any power to our language. We allow the presence of unconditional `goto` statements and `return` statements, as long as they are

used in a structured manner (defined in Section 2.1). A block of code is bracketed by `begin` and `end` statements. Variables can be either numbers (integers, reals, etc.) or arrays of numbers, and expressions include standard arithmetic operations as well as procedure calls.

1.2 Output language

Our output language (an explicitly parallel language) is a parallel extension of our input language. The first extension that we provide is the `doall` construct, which allows us to write parallel loops. A `doall` loop has the same form as a `do` loop, but all of its iterations can be run in parallel. Within a `doall` loop, we will assume that each iteration of the loop effectively receives its own copy of the loop's induction variable.

The other language extension is the `cobegin/coend` construct, which allows us to schedule blocks of statements to run in parallel [AS83]. Each statement or `begin/end` block contained directly within a `cobegin/coend` block can be executed in parallel; we call such a `begin/end` block a parallel block.

1.2.1 Private variables

Our output language has a `private` statement that can be used to specify the generation of private instances of variables within the iterations of a `doall` loop or within a parallel block [IBM88]. Adding `private` statements is similar to renaming [CF87b], but it does not change the variable names in the source program.

When used within a parallel loop, all `private` statements must immediately follow the `doall` statement; when used within a parallel block, all `private` statements must immediately follow the `begin` statement that opens the parallel block. Only non-array variables may be `private` within a parallel loop; both array and non-array variables may be `private` within a parallel block. A `private` statement specifies a variable name and the optional keyword `copyout`; `copyout` means that the value of the private copy

```
(S1)    doall i = 1 to n
(S2)        private x copyout
(S3)        cobegin
(S4)            begin
(S5)                x = 3 * i
(S6)                y(i) = 4 + x
(S7)            end
(S8)            begin
(S9)                private x copyout
(S10)               x = 3 * i
(S11)               z(i) = 5 - x
(S12)           end
(S13)       coend
(S14)   end doall
(S15)   y = 5 * x - 3
```

Figure 1-1: Output language example

of the variable is "copied out" to the surrounding scope's copy of the variable when the `cobegin`/`coend` block or `doall` loop finishes executing. For `doall` loops, the last iteration's private copy is the one whose value is copied out.

When the `copyout` keyword is used with a non-array variable, the value of the variable is copied out; if the `copyout` keyword is used with an array variable, only the elements of the array that are modified in the parallel block are copied out. For a given memory location in a `cobegin`/`coend` block, only one parallel block can have that variable specified as `copyout`; the actual copying occurs when the entire `cobegin`/`coend` block finishes executing. We assume that iteration variables within `doall` loops never require `private` statements.

Figure 1-1 illustrates the use of the `private` statement. S_2 specifies that `x` in S_5 and S_6 effectively refers to an iteration-private copy of `x`. Each iteration can be run in parallel, since no two iterations refer to the same storage location. The `copyout` keyword in S_2 specifies that the value of the last iteration's copy of `x` (the copy of the iteration

where `i` = `n`) is copied into the "global" copy of `x`, which is the copy that S_{15} references.

The references to `x` in S_{10} and S_{11} do not refer to the iteration-private copies of `x`; S_9 specifies that when the `begin/end` block from S_8 to S_{12} is entered, the block effectively receives its own copy of `x`. This block can run in parallel with the other `begin/end` block in the loop, since the two blocks do not refer to the same storage. The `copyout` keyword in S_9 specifies that when the `cobegin/coend` block ends, the value of the copy of `x` within the second `begin/end` block is copied into the iteration-private copy of `x`; this ensures that the last iteration's copy has the correct value when it gets copied out to the global copy.

Making variables `private` statements allows us to increase the parallelism that we can generate: we can run blocks of code (or the iterations of a loop) in parallel if they do not refer to the same storage location. We call the process of making variables `private` *privatization*; in Section 4.1, we give an algorithm to perform privatization [BCFH87].

1.2.2 Semaphores

Our output language has three semaphore statements that allow us to insert explicit synchronization: `semaphore`, `wait`, and `signal`. The `semaphore` statement is used to declare binary semaphores; `wait` and `signal` are the standard semaphore operations (**P** and **V**) [AS83]. Given the sequential program in Figure 1-2(a) as input, we could generate the parallel program in Figure 1-2(b) as output. Since we have inserted `wait` and `signal` statements that refer to the semaphore `s1`, the two `if` blocks can be run in parallel. In Section 4.2, we show how to insert explicit synchronization in our programs.

1.3 Summary

This thesis presents a new representation for sequential programs, the *constrained forward control dependence graph*, from which it is straightforward to generate parallel code. We use control dependence and data dependence information to compute the constrained

```
(S1)   y = 5
(S2)   if p
(S3)       then begin
(S4)           x = 4
(S5)           y = 5 * x + 3
(S6)           z = 6 + x * 5
(S7)       end
(S8)   if q
(S9)       then begin
(S10)          a = 20
(S11)          b = 30 - 5 * a
(S12)          c = 48 * y
(S13)      end
```

Figure 1-2(a): Input program

```
(S1)   y = 5
       cobegin
           semaphore s1
(S2)       if p
(S3)           then begin
(S4)               x = 4
(S5)               y = 5 * x + 3
                   signal s1
(S6)               z = 6 + x * 5
(S7)           end
               else signal s1
(S8)       if q
(S9)           then begin
(S10)              a = 20
(S11)              b = 30 - 5 * a
                   wait s1
(S12)              c = 48 * y
(S13)          end
       coend
```

Figure 1-2(b): Output program

Figure 1-2: Semaphore example

forward control dependence graph: control dependences describe syntactic constraints on parallelism, while data dependences describe semantic constraints. In addition, we present two algorithms that utilize the constrained forward control dependence graph: an algorithm to perform privatization, and an algorithm to add explicit synchronization.

Figure 1-3 illustrates the types of transformations that we want to perform: given the sequential program in Figure 1-3(a) as input, we want to generate the parallel program in Figure 1-3(b) as output. Since the first two `do` loops in Figure 1-3(a) initialize two separate arrays, we can run them in parallel with each other. Since each iteration of these two loops writes to different elements of `X` and `Y`, respectively, we can run the iterations of each loop in parallel. Since each statement within these loops also writes to different elements of `X` and `Y`, they also can be run in parallel. However, since the final `do` loop uses the values of `X` and `Y` that are set in the first two loops, it cannot be executed until the first two loops both finish.[1] When the final `do` loop does execute, all of its iterations can be run in parallel, since they all write to different elements of `Z`.

Our parallelization techniques are described for single procedures (or programs); however, they work in an interprocedural setting as well, since code can still be generated on a procedure-by-procedure basis. Interprocedural data flow analysis is necessary to generate *good* parallel code; without it, worst-case assumptions about data dependence must be made about procedure calls.

In Chapter 2, we describe the program analyses that we need to compute the constrained forward control dependence graph; in Chapter 3, we describe the constrained forward control dependence graph; and in Chapter 4, we present algorithms to perform privatization, to insert explicit synchronization, and to generate parallel code from our representation. The new work in this thesis is the material described in Chapters 3 and 4.

[1] We schedule all parallelism at compile-time; a dataflow machine could begin to execute the final `do` loop before the other two loops finish, as could a pipelined machine. The addition of explicit synchronization could also allow more parallelism.

```
(S1)   do i = 1 to n
(S2)       X(2*i) = i + 5
(S3)       X(2*i+1) = 3 * i
(S4)   end do
(S5)   do j = 1 to n
(S6)       Y(2*i) = i - 4
(S7)       Y(2*i+1) = i * i
(S8)   end do
(S9)   do k = 2 to 2 * n + 1
(S10)      Z(i) = X(i) + Y(i)
(S11)  end do
```

Figure 1-3(a): Input program

```
       cobegin
(S1)       doall i = 1 to n
               cobegin
(S2)               X(2*i) = i + 5
(S3)               X(2*i+1) = 3 * i
               coend
(S4)       end doall
(S5)       doall j = 1 to n
               cobegin
(S6)               Y(2*i) = i - 4
(S7)               Y(2*i+1) = i * i
               coend
(S8)       end doall
       coend
(S9)   doall k = 1 to 2 * n + 1
(S10)      Z(i) = X(i) + Y(i)
(S11)  end doall
```

Figure 1-3(b): Output program

Figure 1-3: Parallelization example

Chapter 2

Program analyses

This chapter describes the compiler analyses required to construct our program representation.

2.1 Control flow

Definition 1 A *control flow graph* $G_f(P) = (N_f(P), E_f(P))$ of a program P is a directed graph, where the set of nodes $N_f(P)$ contains two distinguished nodes *Entry* and *Exit*, the set $N_f(P) - \{Entry, Exit\}$ is isomorphic to the statements of P, and the set of edges $E_f(P)$ corresponds to the possible flow of control within P [ASU86]. We assume that all programs are single-entry and single-exit without loss of generality [Hec77].

Figure 2-2 contains the control flow graph for the program in Figure 2-1; it illustrates the form of the control flow subgraph for a `do` loop. The `do` statement branches around the body of the loop because it represents the initial test of the iteration variable against the loop bounds. The `end do` statement represents an "increment and test" statement, which branches to the "header" of the loop when the iteration variable is less than the upper loop bound.

A node n *dominates* a node m if all paths from *Entry* to m in $G_f(P)$ contain n; n *post-dominates* m if all paths from m to *Exit* in $G_f(P)$ contain n. A node n *properly*

```
(S1)   if p
(S2)       then do i = 1 to n
(S3)           if q
(S4)               then begin
(S5)                   a = 10 + i
(S6)                   X(i) = 5 - i
(S7)               end
(S8)           Y(i) = 4 * i
(S9)           Z(i) = X(i-1) + 5
(S10)      end do
(S11)      else begin
(S12)          a = 4
(S13)          b = 3 * a
(S14)      end
```

Figure 2-1: Sample program

post-dominates a node m if n post-dominates m and $n \neq m$ [Hec77]; the *immediate post-dominator* of a node m is the proper post-dominator n of m such that every proper post-dominator of m post-dominates n.

Proper post-domination specifies how we can schedule statements to execute. In the control flow graph in Figure 2-2, S_{10} properly post-dominates all of the statements in the set $\{S_3, S_4, \ldots, S_9\}$; if any of those statements execute, S_{10} will be executed later. We will schedule a statement n to run in parallel with another statement m only if n properly post-dominates m; since n will execute each time m executes, we are guaranteed to accomplish some work in parallel.

Our assumption that all loops are `do` loops results in a loss of generality. However, many non-`do` loops formed from `goto` and `if-then-else` statements can be converted into `do` loops by performing induction variable analysis [Lea85]. The parallelization of arbitrary *while* or *until* loops is beyond the scope of this thesis.

We assume that every program is *structured* [LM77, Mil82, BJ66]. Structured programs are a class of programs whose control flow graphs can be generated from a family

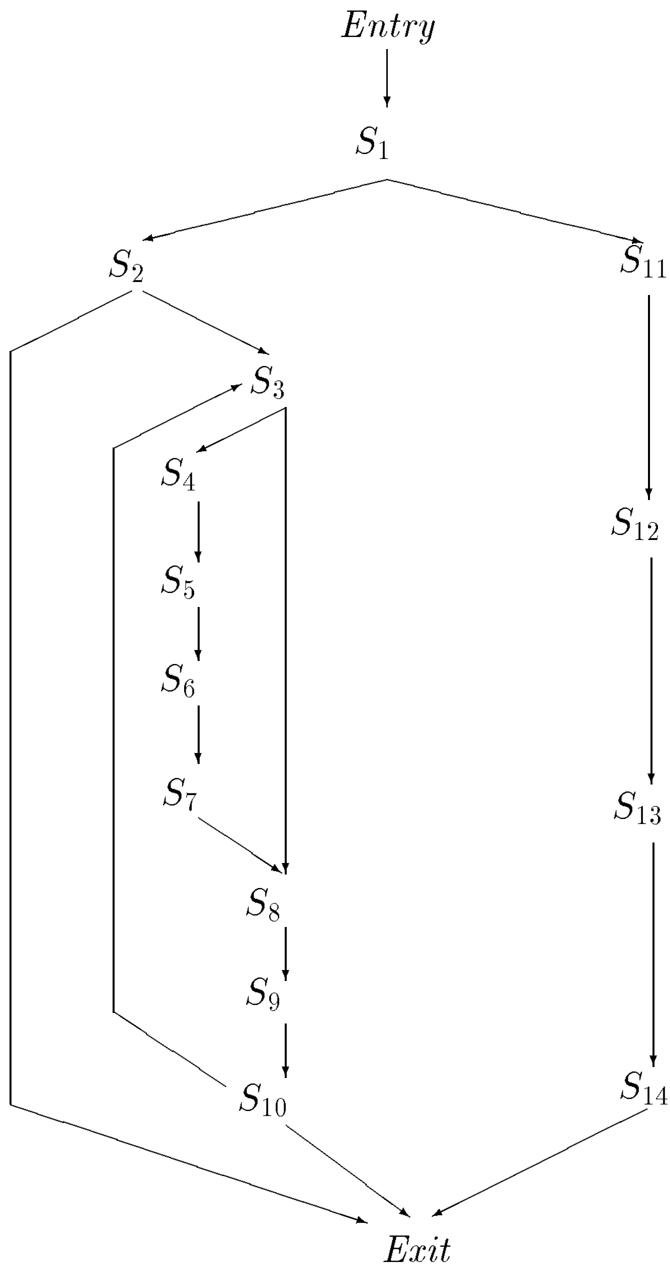

Figure 2-2: Control flow graph of program in Figure 2-1

of simple graph grammars whose productions represent sequencing, conditionals, and looping; Figure 2-3 contains a graph grammar that can generate our programs. Any non-structured program is equivalent to some structured program, so the assumption of structuredness loses no generality.

The property of being structured implies a property called *reducibility*; if a program P is reducible, any strongly-connected region (i.e. loop) in $G_f(P)$ contains a single node (called the *header* of the region) that dominates all of the nodes in the region, which means that the header is the first node executed in the region [ASU86]. Since any irreducible program can be transformed into a reducible program via *node splitting*, the assumption of reducibility alone also loses no generality [Hec77].

We define an edge $e \in E_f(P)$ to be a *back edge* of $G_f(P)$ if its head dominates its tail in $G_f(P)$.[1] Reducibility is a useful assumption because the non-back edges in a reducible control flow graph form a DAG (directed acyclic graph) [ASU86].

2.2 Control dependence

We use a form of *control dependence* to represent the parallelism that we can extract from a program [Ban79, FOW87]. Several algorithms for computing the control dependence relation are known [CF87a, FOW87].

Definition 2 Given a control flow graph $G_f(P)$, a node m is *control dependent* upon a node n via a control flow edge e_f (alternatively, n *controls* m via e_f) if all of the following conditions hold:

- There exists a path from n to m in $G_f(P)$.
- m does not properly post-dominate n in $G_f(P)$.
- m post-dominates n' in $G_f(P)$, where $e_f = (n, n') \in E_f(P)$.

[1] The head of an edge $e = (m, n)$ is n; the tail is m.

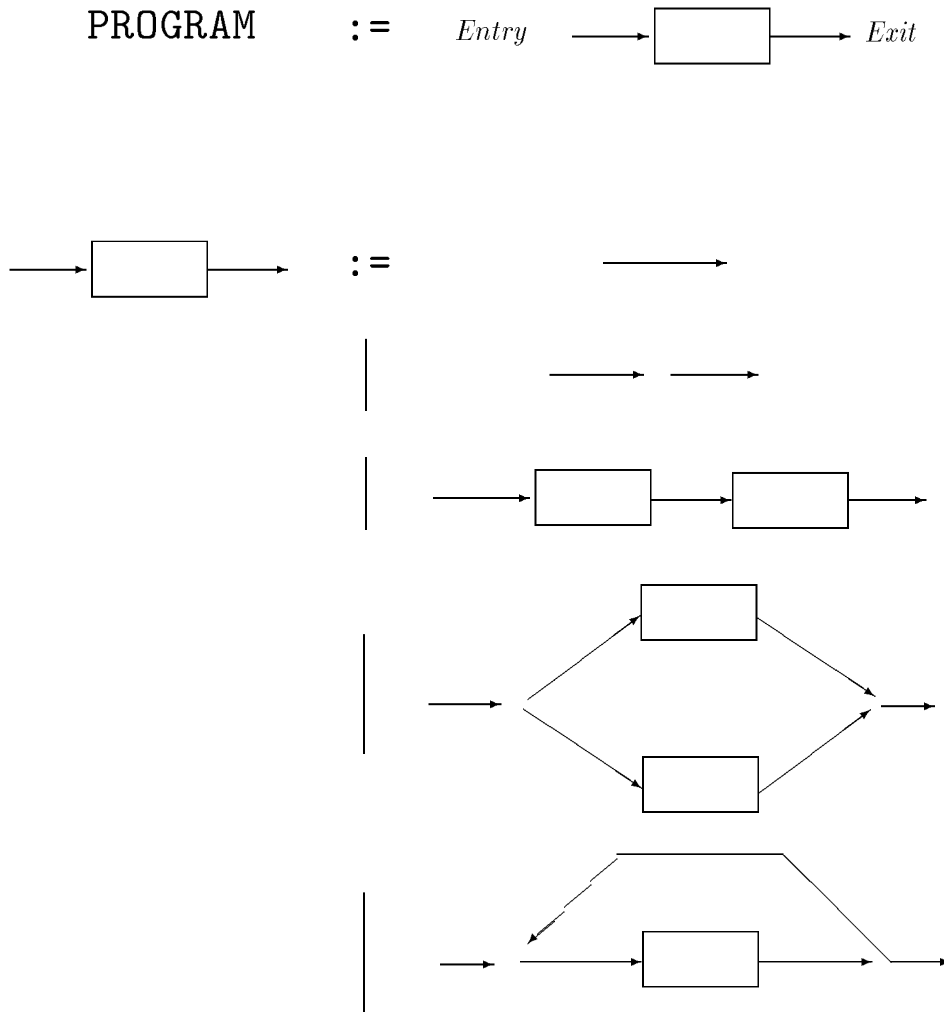

Figure 2-3: Graph grammar for structured programs

If a node n controls a node m via a control flow edge e_f, then exiting n on e_f will always result in m executing; exiting n on another edge may or may not result in m executing. In other words, whether m executes depends on whether the flow of control leaves n via e_f.

We define a *control dependence graph* $G_c(G_f) = (N_c(G_f), E_c(G_f))$, where $N_c(G_f) = N_f$. The edges in the set E_c are labeled edges, where a labeled edge $e_c = ((n, m), e_f)$ is in $E_c(G_f)$ if and only if n controls m via edge e_f in G_f. A node with children in a control dependence graph is a decision point where a choice between control flow edges will be taken.

The control dependence graph for the program in Figure 2-1 (page 19) is shown in Figure 2-4; the labels of the control dependence edges are omitted for clarity. Assume that the edges of the control flow graph in Figure 2-2 are named as follows: $e_a = (S_1, S_2)$, $e_b = (S_1, S_{11})$, $e_c = (S_2, S_3)$, $e_d = (S_3, S_4)$, and $e_e = (S_{10}, S_3)$. The labels of the control dependence edges in Figure 2-4 are:

- The label of the edge (S_1,S_2) is e_a, and the labels of the edges (S_1,S_{11}) (S_1,S_{12}), (S_1,S_{13}), and (S_1,S_{14}) are e_b.
- The labels of the edges (S_2,S_3), (S_2,S_8), (S_2,S_9), and (S_2,S_{10}) are e_c.
- The labels of the edges (S_3,S_4), (S_3,S_5), (S_3,S_6), and (S_3,S_7) are e_d.
- The labels of the edges (S_{10},S_3), (S_{10},S_8), (S_{10},S_9), and (S_{10},S_{10}) are e_e.

Lemma 1 If there exist two distinct nodes m_0 and m_1 that are both control dependent upon the same node n via the same control flow edge e_f, then exactly one of the following is true:

- m_0 properly post-dominates m_1.
- m_1 properly post-dominates m_0.

Proof: By the definition of control dependence, both m_0 and m_1 post-dominate the head of e_f.

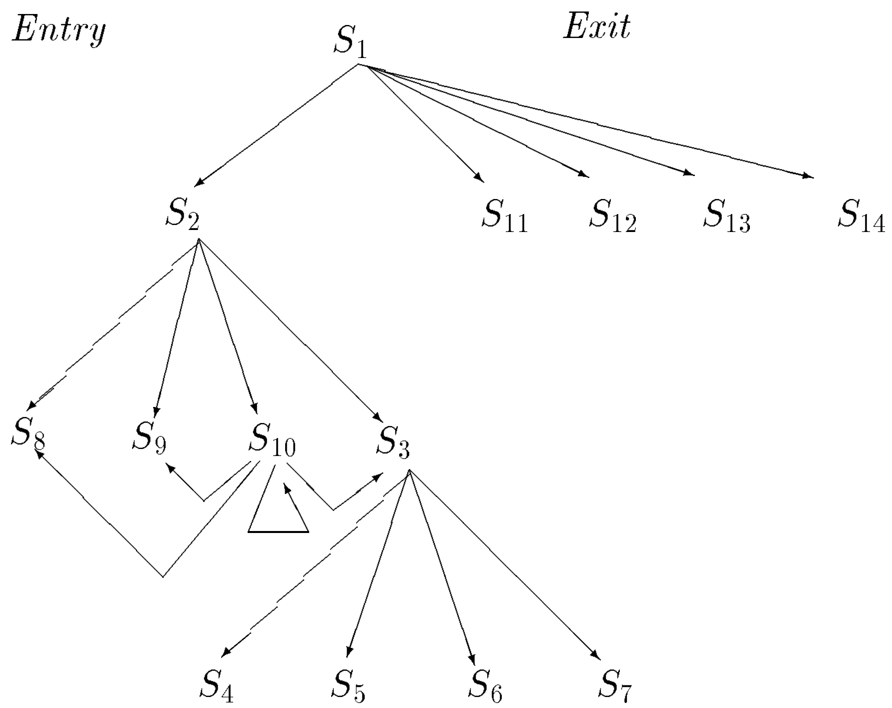

Figure 2-4: Control dependence graph of program in Figure 2-1

The control dependence graph represents constraints on the parallelism that we can extract from a program. If a node m is controlled by a node n, we cannot run m in parallel with n, since whether m executes depends on how the flow of control leaves n. However, by Lemma 1, we can possibly execute two nodes in parallel if they are siblings in the control dependence graph: if two nodes m and m' are both controlled by n via e_f, both will always be executed if we exit n via e_f. In addition, if m and m' both have children in the control dependence graph, we can possibly schedule the two groups of children to run in parallel with each other.

The control dependence graph in Figure 2-4 specifies which statements we can run in parallel:

- After S_1 executes, either we will execute S_2 and possibly the statements that it indirectly controls, or we will execute S_{11}, S_{12}, S_{13}, and S_{14}, all of which we could run in parallel.

- After S_2 executes, we may execute S_3, S_8, S_9, and S_{10}, all of which could run in parallel.

- After S_3 executes, we may execute S_4, S_5, S_6, and S_7, all of which could run in parallel.

Although the control dependence graph specifies which statements we can run in parallel due to the syntactic structure of a program, it does not take into account the semantics of the program: in the program in Figure 2-1 (page 19), we cannot execute S_{12} and S_{13} in parallel, since S_{13} makes use of the value that S_{12} computes. We use a *data dependence graph* (described in Section 2.3) to compute semantic constraints on parallelism.

Since the control dependence graph is very unstructured, generating code from it is difficult. In general, a control dependence graph will contain loops: in Figure 2-4, S_{10} controls itself and every other statement within the loop that post-dominates S_3. However, we do not need such information: it complicates code generation, and we do

not need it to generate `doall` loops. The control dependence graph is also in general unconnected, since any node that post-dominates *Entry* will not be control dependent upon *Entry*: in Figure 2-4 both S_1 and *Exit* are not controlled by any node. This complicates code generation, since we cannot make a simple pass through the graph. Our program representation, the *constrained forward control dependence graph* (described in Section 3.1), does not have these problems.

Definition 3 Given a control dependence graph $G_c(G_f)$, a node m is *indirectly control dependent* upon a node n via control flow edge e_f (alternatively, n *indirectly controls* m via e_f) if all of the following conditions hold:

- There exists a path p from n to m in $G_c(G_f)$.
- The first control dependence edge in p has label e_f.

If two nodes m_1 and m_2 are indirectly control dependent on the same node n via the same control flow edge e_f, they may be scheduled to run in parallel (if one node is an ancestor of the other, the control dependence graph constrains the parallelism; if neither node is an ancestor of the other, they could be scheduled to run in parallel). If one node must run only after the other, some of the common indirect controllers of m_1 and m_2 must be "marked" to ensure that m_1 and m_2 are not scheduled to run in parallel. We describe which common indirect controllers must be marked in Section 3.2.

2.3 Data dependence

Data dependences are constraints on the order of statement execution in sequential programs that are due to either the flow of values or the reuse of storage; they must be taken into account to ensure that the semantics of the output code is correct. Following Kuck's notation, there are three types of data dependence: *flow dependence*, *anti dependence*, and *output dependence* [Kuc78, PW86].

If a statement S_i is *data dependent* upon another statement S_j in a sequential program, it must be executed after S_j. We modify Kuck's definitions to deal with arrays more precisely (numeric variables could be considered as single-element arrays) [Kuc78]:

- S_i is *flow dependent* upon S_j over certain elements of a variable v if an instance of S_j assigns values to those elements of v and an instance of S_i may use those values of those elements of v.

- S_i is *anti dependent* upon S_j over certain elements of a variable v if an instance of S_i assigns values to those element of v and an instance of S_j uses those elements of v, but not the values assigned by the instance of S_i.

- S_i is *output dependent* upon S_j over certain elements of a variable v if instances of S_i and S_j both assign values to those elements of v, and the values that the instance of S_i assigns must be stored after the values that the instance of S_j assigns.

The last two types of data dependence are *storage-related*, in that they are due to the reuse of storage locations (variables) in a program.

There is a flow dependence from S_1 to S_3 in the program in Figure 2-5, since the execution of S_3 on the first iteration of the `do` loop will use the value of `X(1)`. Similarly, there is a flow dependence from S_3 to itself, since the execution of S_3 on some iteration could use the value defined in the previous iteration by S_3; there is a flow dependence from S_5 to S_3 by the same argument. There is a flow dependence from S_3 to S_5, since S_3 defines `X(i)` and S_5 may use `X(i)` on any iteration; similarly, there is an output dependence from S_3 to S_5 since both statements could define `X(i)` on any iteration.

There is no output dependence from S_3 to itself, because each iteration of the loop assigns to a different element of `X`; similarly, there is no output dependence from S_5 to itself.

We further classify data dependences as *loop-carried* or *loop-independent* [All83]. A data dependence is loop-carried if the data dependence goes from a statement in one

```
(S1)   X(1) = 5
(S2)   do i = 2 to n
(S3)       X(i) = 4 * X(i-1) + 2
(S4)       if p
(S5)           then X(i) = X(i) * 4
(S6)   end do
```

Figure 2-5: Data dependence example

```
(S1)   do i = 1 to n
(S2)       if p
(S3)           then x = 3
(S4)       if q
(S5)           then x = 5
(S6)   end do
```

Figure 2-6: Loop-independent and loop-carried dependences

iteration of a loop to a statement in a later iteration of the same loop; otherwise, the data dependence is loop-independent. In the program in Figure 2-5, the flow dependence from S_1 to S_3 is loop-independent, since the dependence does not cross between the iterations of any loop. The flow dependence from S_3 to S_5 and the output dependence from S_3 to S_5 are also loop-independent, since they occur within a single iteration of the `do` loop. However, the flow dependences from S_3 to itself and from S_5 to S_3 are loop-carried, since the dependences go from one iteration of the `do` loop to the next.

In the program in Figure 2-6, there are two data dependences present from S_3 to S_5: both a loop-independent output dependence and a loop-carried output dependence. There is a loop-independent output dependence since both statements may define `x` within any iteration of the `do` loop; there is a loop-carried output dependence since the definition of `x` by S_5 in some iteration of the loop could overwrite the value of `x` defined by S_3 in a previous iteration. There is a loop-carried output dependence from S_5 to S_3,

since S_3 in some iteration could overwrite the value of `x` defined by S_5 in a previous iteration; there is a loop-carried output dependence from S_3 to itself, since S_3 in some iteration could overwrite the value of `x` defined by S_3 in a previous iteration; and there is a loop-carried output dependence from S_5 to itself, since S_5 in some iteration could overwrite the value of `x` defined by S_5 in a previous iteration.

Data dependences restrict the parallelism we can generate; the classification of data dependences as loop-carried or loop-independent is important because it parallels the distinction between the two types of parallelism that we generate: parallel loops and parallel blocks. Intuitively, loop-carried data dependences restrict the parallelism we can achieve among the iterations of a loop, and loop-independent data dependences restrict the parallelism we can achieve from parallel blocks.

We can use interprocedural data flow analysis [Bur87, Hec77] and data dependence analysis [Cyt87, Wol82] to find the data dependences that exist in a program, which we represent via a *data dependence graph* $G_d(P) = (N_d(P), E_d(P))$, where $N_d(P) = N_f(P)$, and $E_d(P)$ is a set of labeled edges called *data dependence edges*.[2] Data dependence edges are of the form $((n_1, n_2), v, t, l)$, where v is a variable, t is either **flow**, **anti**, or **output**, and l is either **independent** or $(\mathbf{carried}, n_{do})$; $n_{do} \in N_f(P)$ is the `do` node that corresponds to the loop that carries the dependence. Data dependence edges thus represent all six types of data dependence: loop-carried flow, anti, and output dependences, and loop-independent flow, anti, and output dependences.

2.3.1 Data flow analysis

We need the solution to two distributive data flow problems for our algorithms: the reaching definitions and live variable problems. The solutions to these two problems are in a class known as "meet over all paths" solutions [Hec77]. Solutions to data flow problems are inherently conservative, since they are based on compile-time knowledge.

[2]Standard interprocedural data flow analysis and data dependence analysis will not give the exact information about arrays that we desire; further research is necessary to determine how to compute such information.

A definition is a pair (n, v), where n is a node and v is a storage location (numeric variable or array element) that n defines. A definition (n, v) *reaches* another node m if that definition is "available" at m. A storage location v is *live* at a node n if the value of v immediately before n executes could be used when or after n executes; if v is not live at n, it is *dead* at n.[3]

[3]These definitions are more precise with respect to arrays than standard definitions; problems in computing the solutions are similar to the problems in computing our data dependence graph.

Chapter 3

Constrained forward control dependence

We use *forward control dependence* to represent the parallelism that we can extract from a program. Control dependence does not give us the exact information that we want: the control dependence graph is unconnected and has too much information about loops, in that the exit from a loop controls other statements in the loop. We discard this "extra" information about loops since it is not useful for our transformations.

To remove the extra control dependence edges and to create a connected graph, we define a *forward control flow graph*, from which we compute the *forward control dependence graph*. The forward control flow graph contains all of the non-back edges of the control flow graph, and an additional edge from *Entry* to *Exit*.

Definition 4 Given a reducible control flow graph $G_f(P)$, the *forward control flow graph* $G_{ff}(P) = (N_{ff}(P), E_{ff}(P))$ is defined as follows, where $Forward(E_f(P))$ contains exactly the non-back edges in $E_f(P)$:

$$\begin{aligned} N_{ff}(P) &= N_f(P) \\ E_{ff}(P) &= Forward(E_f(P)) \cup (Entry, Exit) \end{aligned}$$

$G_{ff}(P)$ is a DAG, since the non-back edges of a reducible control flow graph form a DAG

and the edge $(Entry, Exit)$ cannot create a cycle.

Figure 3-1 shows the forward control flow graph of the program in Figure 2-1 (page 19); compare it with the control flow graph in Figure 2-2 (page 20). The only node that post-dominates *Entry* in the forward control flow graph is *Exit*, and there are no strongly-connected regions in the forward control flow graph.

3.1 Forward control dependence

The forward control dependence graph is simply the control dependence graph of the forward control flow graph minus *Exit*.

Definition 5 The *forward control dependence graph* (alternately, the f-control dependence graph) $G_{fc}(P) = (N_{fc}(P), E_{fc}(P))$ of a program P is defined as follows:

$$\begin{aligned} N_{fc}(P) &= N_c(G_{ff}(P)) - Exit \\ E_{fc}(P) &= E_c(G_{ff}(P)) \end{aligned}$$

Since *Exit* properly post-dominates every other node in $G_{ff}(P)$, there are no edges to *Exit* in $G_c(G_{ff}(P))$, and $G_{fc}(P)$ is a well-defined graph.

Figure 3-2 contains the forward control dependence graph of the program in Figure 2-1 (page 19); compare it with the control dependence graph in Figure 2-4 (page 24). There are no outgoing edges from S_{10}, there is an edge $(Entry, S_1)$, and there are no loops in the f-control dependence graph, which is connected.

3.1.1 Discussion

Theorem 1 An f-control dependence graph $G_{fc}(P)$ is connected, and the distinguished node *Entry* is its root.

Proof: First, we show that any node in $G_{fc}(P)$ is reachable from *Entry* in $G_{fc}(P)$. Take any node $n \in N_{fc}(P)$, $n \neq Entry$, and take any path $p = (Entry, p_1, p_2, \ldots, p_k, n)$ from

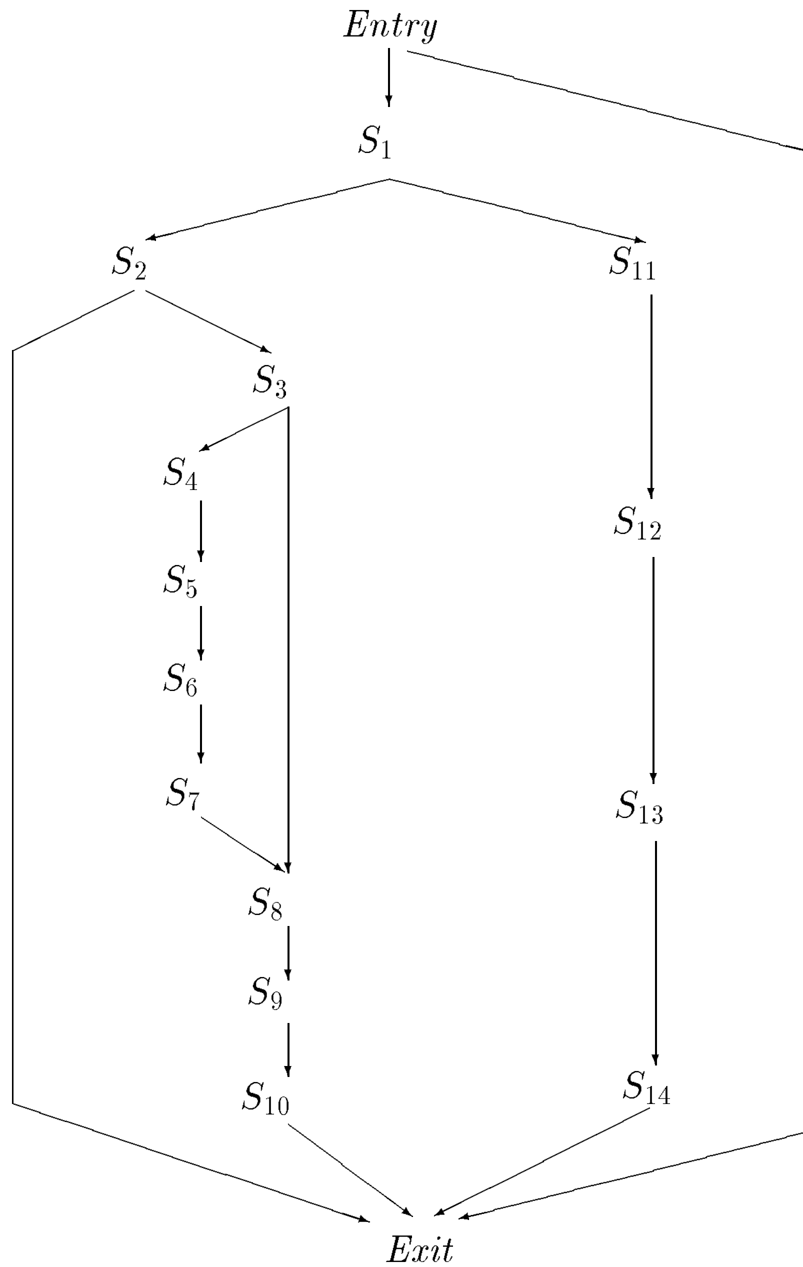

Figure 3-1: Forward control flow graph of program in Figure 2-1

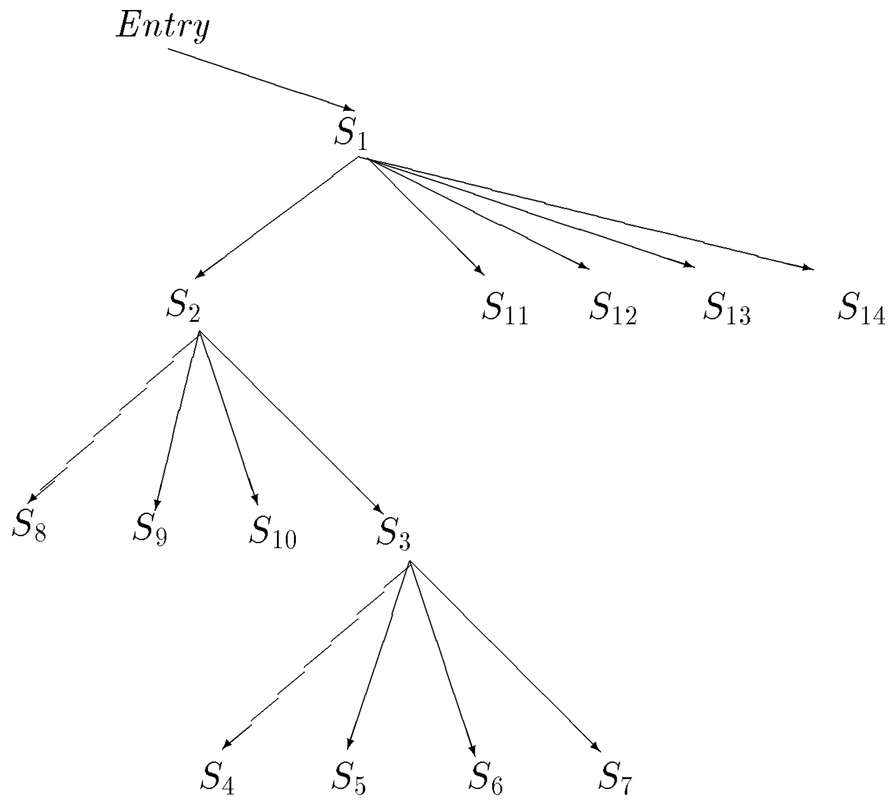

Figure 3-2: Forward control dependence graph of program in Figure 2-1

Entry to n in $G_{ff}(P)$. We prove by induction on the length l of p that there is a path from *Entry* to n in $G_{fc}(P)$.

If the length of p is one, *Entry* controls n, since n cannot post-dominate *Entry* in $G_{ff}(P)$.

If the length of p is $l+1$, then n either does or does not post-dominate p_k. If n does not post-dominate p_k, p_k controls n. By the induction hypothesis, there is a path from *Entry* to p_k in $G_{fc}(P)$, which implies there is a path from *Entry* to n in $G_{fc}(P)$. If n does post-dominate p_k, it must be controlled by the node that controls p_k. By the induction hypothesis, there is a path from *Entry* to p_k in $G_{fc}(P)$; there must be a path from *Entry* to p_k's parent in $G_{fc}(P)$, which implies that there is a path from *Entry* to n in $G_{fc}(P)$.

Finally, no node can control *Entry*, so *Entry* is the root.

Since the f-control dependence graph is connected, generating code is simple - a single pass through the graph starting at *Entry* is guaranteed to visit every node.

Theorem 2 An f-control dependence graph $G_{fc}(P)$ is a DAG (directed acyclic graph); it is a tree if p is structured.

Proof: By induction on the graph grammar for structured programs (Figure 2-3 on page 22).

Since we assume that all programs are structured, all f-control dependence graphs will be trees. The f-control dependence graph directly represents the nested parallelism that we generate: at each level in the tree we schedule siblings and their subtrees to run in parallel, subject to constraints due to data dependences (discussed in Section 3.2).

Theorem 3 Given a structured program whose loops are `do` loops, the forward control dependence graph $G_{fc}(P)$ will have the following property: a `do` node d_i of a `do` loop is the root of a subtree $S_i \subset G_{fc}(P)$ such that S_i contains exactly the nodes in the `do` loop.

Proof: Directly from graph of `do` loop.

If we perform a top-down traversal of the f-control dependence graph to generate code, we visit a `do` node before we visit the nodes in its loop. Thus, the `do` node of a loop provides a convenient place to store information regarding the parallelization of the loop.

3.2 Data dependence constraints

We add *data dependence constraints* to the forward control dependence graph to ensure that we preserve the semantics imposed by data dependences in the program. These constraints are of two types: constraint edges and constraint marks. Edges represent restrictions on *cobegin*-type parallelism; marks represent restrictions upon *doall*-type parallelism. Algorithms for computing data dependence constraints are given in Appendix A.

3.2.1 Constraint edges

Given a loop-independent data dependence from a node n to a node m, we must ensure that m executes only after n. For each loop-independent data dependence edge $e_d \in E_d(P)$ from n to m, we add a constraint edge, a labeled edge of the form $((m_i, m_j), LCA(n,m), e_d)$, where $m_i, m_j, LCA(n,m) \in N_{fc}(P)$, and m_i, m_j are children of $LCA(n,m)$ in $G_{fc}(P)$. This constraint edge means that when we generate code for $LCA(n,m)$, we cannot run the subtrees beneath m_i and m_j in parallel.

Lemma 2 Given a loop-independent data dependence edge from a node n to a node m, and $LCA(n,m) \in N_{fc}(P)$, if $LCA(n,m) \neq n$ and $LCA(n,m) \neq m$, then $LCA(n,m)$ indirectly controls both n and m via the same control flow edge e_f.

Proof: Assume that $LCA(n,m) \neq n$, $LCA(n,m) \neq m$, and that $LCA(n,m)$ does not indirectly control n and m via the same control flow edge. Let $LCA(n,m)$ indirectly control n via e_n and m via e_m. If e_n is taken upon leaving $LCA(n,m)$, n will be executed and m will not be executed; if e_m is taken, m will be executed and n will not be executed.

There cannot exist a loop-independent data dependence e_d from n to m; there could only exist a loop-carried data dependence from n to m.

If there is some loop-independent data dependence between two nodes n and m, we indicate this at $LCA(n, m)$ to prevent us from generating code to run n and m in parallel. If $LCA(n, m) = n$ or $LCA(n, m) = m$ we do not need to add a constraint, since the forward control dependence edges already order n and m.

Lemma 3 The set of constraint edges is acyclic.

Proof: Take two siblings n and m in $N_{fc}(P)$ that are controlled via the same control flow edge e_f. Either n properly post-dominates m, or m properly post-dominates n. Without loss of generality, assume that m properly post-dominates n; this implies that m also properly post-dominates all of n's descendants in $G_{fc}(P)$. Take some depth-first numbering *dfs#* of the nodes in $G_f(P)$. Let n' be a descendant of n and m' a descendant of m. Since m properly post-dominates n', *dfs#*(n') < *dfs#*(m); since there is a path in $G_f(P)$ from m to m', *dfs#*(m) < *dfs#*(m'). Therefore, *dfs#*(n') < *dfs#*(m').

By definition, the head of a loop-independent data dependence edge must have a greater *dfs#* than the tail; this implies that data dependence edges can only go from descendants of n to descendants of m. Therefore, constraint edges can only go in one direction, from n to m.

Since the proper post-domination relation totally orders all siblings, the set of constraint edges is acyclic.

A constraint edge both groups and orders siblings in $G_{fc}(P)$. Each set of siblings that is connected by constraint edges must be generated as part of the same parallel block; siblings that are connected by constraint edges cannot be members of different parallel blocks since they may refer to the same storage location. Within each parallel block, the constraint edges specify the order in which we must generate the nodes.

3.2.2 Constraint marks

Given a loop-carried data dependence edge from n to m, we must ensure that the loop that carries the dependence is not transformed into a `doall` loop. We mark this information at the `do` node of the loop, since our code generation pass will always visit the `do` node before visiting any nodes in the body of the loop (Theorem 3). For each loop-carried data dependence in $G_d(P)$ we add a constraint mark, which is of the form (n_{do}, e_d), where $n_{do} \in G_{fc}(P)$ is a `do` node, and e_d is a loop-carried data dependence edge.

During code generation, the presence of a constraint mark (n_{do}, e_d) indicates that n_{do}'s loop (the `do` loop that corresponds to n_{do}) cannot be run as a *doall* loop, since e_d goes between some of its iterations.

3.2.3 Discussion

The constraints on the f-control dependence graph represent sequentialization that we must obey during code generation. The f-control dependence graph of the program in Figure 3-3(a) is shown in Figure 3-4. The following data dependences edges exist in the program:

- There is a loop-independent flow dependence edge e_{d1} from S_6 to S_9.
- There is a loop-independent flow dependence edge e_{d2} from S_{12} to S_{13}.
- There is a loop-carried output dependence edge e_{d3} from S_5 to itself.
- There is a loop-carried output dependence edge e_{d4} from S_6 to itself.

There are no data dependences between statements in the set $\{S_5, S_6, S_8, S_9\}$ and the set $\{S_{12}, S_{13}\}$; every control flow path from *Entry* to *Exit* contains either the first set of statements or the second, but not both.

The two flow dependences correspond to two constraint edges: $e_1 = ((S_3, S_9), S_1, e_{d1})$ and $e_2 = ((S_{12}, S_{13}), S_1, e_{d2})$. The two output dependences correspond to two constraint marks: $m_1 = (S_4, e_{d3})$ and $m_2 = (S_4, e_{d4})$. We can generate code that obeys the con-

```
(S1)    if p
(S2)        then begin
(S3)            if q
(S4)                then do i = 1 to n
(S5)                    a = 10 * i
(S6)                    x = 5 - i
(S7)                end do
(S8)            y = 4
(S9)            z = x + 5
(S10)       end
(S11)       else begin
(S12)           z = 4
(S13)           y = 3 * z
(S14)       end
```

Figure 3-3(a): Input program

```
(S1)    if p
(S2)        then begin
                cobegin
(S3)                if q
(S4)                    then do i = 1 to n
                            cobegin
(S5)                            a = 10 * i
(S6)                            x = 5 - i
                            coend
(S7)                    end do
(S8)                y = 4
                coend
(S9)                z = x + 5
(S10)       end
(S11)       else begin
(S12)               z = 4
(S13)               y = 3 * z
(S14)       end
```

Figure 3-3(b): Output program

Figure 3-3: Constraint edge discussion

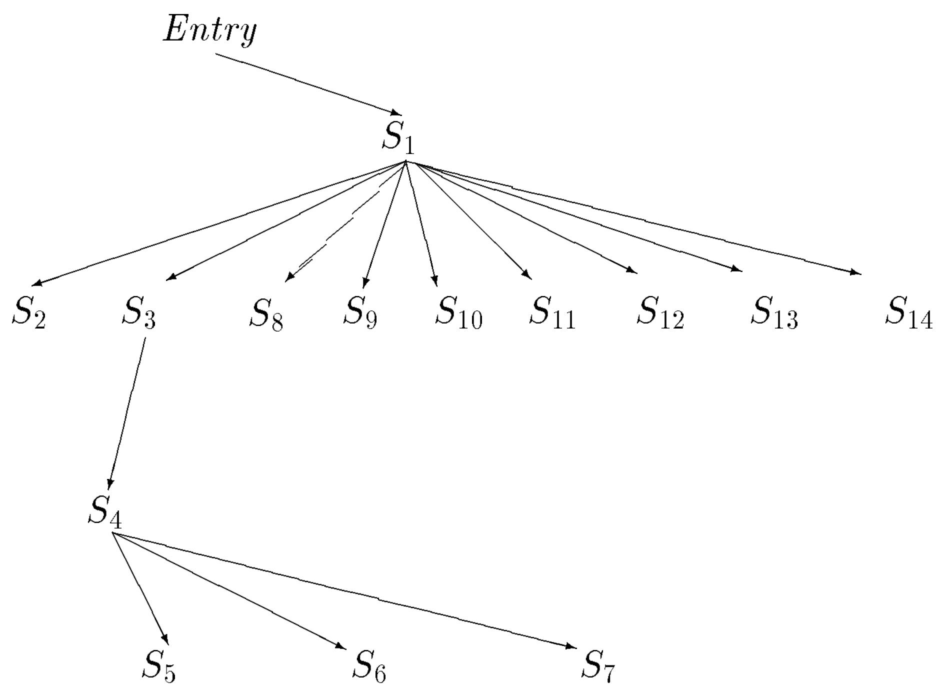

Figure 3-4: Forward control dependence graph of program in Figure 3-3(a)

straints in a straightforward manner. We generate sequential code such that the forward control dependence subtree beneath S_9 executes after the subtree beneath S_3, and such that the subtree beneath S_{13} executes after the subtree beneath S_{12}; this satifies the constraint edges. We do not transform the `do` loop into a `doall` loop; this satisfies the constraint marks. The output code that we would produce is shown in Figure 3-3(b).

The constraints thus represent implicit synchronization - each one means that we must sequentialize certain blocks of code or loop iterations before other blocks or loop iterations. This does not give us good performance for two reasons: the original program may have reused variable names, thus creating storage-related data dependences that can be eliminated; and we sequentialize entire blocks of code, parts of which could be run in parallel.[1] In Chapter 4 we present algorithms to deal with these problems: in Section 4.1 we present an algorithm to perform privatization, a program optimization that removes certain storage-related data dependences; in Section 4.2 we present an algorithm to remove certain data dependence constraints and insert explicit synchronization (`wait` and `signal` semaphore commands).

[1] In Figure 3-3(b), the `do` loop could be a `doall` loop if each iteration of the loop received private copies of the variables `x` and `a`.

Chapter 4

Algorithms

This chapter describes algorithms for inserting synchronization, performing privatization, and generating code. All of these algorithms utilize the constrained forward control dependence graph.

4.1 Privatization

We use a technique that we call *privatization* to increase the amount of parallelism in a program: it removes some storage-related constraint edges (those due to anti and output data dependences) from a program [BCFH87].

In the program in Figure 4-1(a), we would like to execute statements S_1 and S_2 in parallel with statements S_3 and S_4; however, there are storage-related data dependences between the statements that prevent us from doing so. The program reuses the storage for `x`: there is an anti dependence from S_2 to S_3 and an output dependence from S_1 to S_3, which correspond to constraint edges between the same nodes. Privatization increases the parallelism by transforming the program into two parallel processes (one consisting of S_1 and S_2, and one consisting of S_3 and S_4), where each process has its own *private* copy of `x`; Figure 4-1(b) shows the result of privatizing `x`. We distinguish between two types of privatization: block privatization, which increases `cobegin`-type parallelism, and loop

```
(S1)    x = 3
(S2)    y = x
(S3)    x = 5
(S4)    z = x
```

Figure 4-1(a): Input program

```
        cobegin
            begin
(S1)            x = 3
(S2)            y = x
            end
            begin
                private x copyout
(S3)            x = 5
(S4)            z = x
            end
        coend
```

Figure 4-1(b): Output program

Figure 4-1: Privatization example

```
(S1)   x = 10
(S2)   if p
(S3)       then x = x + 5
(S4)   y = 3 * x
```

Figure 4-2(a): Sample program

```
(S1)   x1 = 10
(S2)   if p
(S3)       then x2 = x1 + 5
           else x2 = x1
(S4)   y = 3 * x2
```

Figure 4-2(b): Renamed program

Figure 4-2: Privatization versus renaming

privatization, which increases `doall`-type parallelism.

Block privatization is a form of renaming [CF87b]; however, there are some important differences between them. Privatization does not change any variable names; the variable names from the sequential program are preserved, which makes output programs more readable. Also, block privatization only renames variables in a manner consistent with the parallelism that we generate; since we only run siblings and their subtrees to run in parallel, privatization does not transform every variable that could be renamed. In the program in Figure 4-2(a), instances of the variable `x` could be renamed to get the program in Figure 4-2(b). However, privatization does not transform any variables in this example; we do not generate code to execute S_1 and S_3 in parallel, so we do not rename the instances of `x` in them.

Loop privatization is similar to scalar expansion; each reference to a private variable in a loop iteration essentially refers to a member of a vector of private variables [PW86]. Loop privatization also preserves variable names.

```
(S1)   z = 25
(S2)   x = 3 + z * 2
(S3)   y = 44 * x
(S4)   z = 32 * y + 19
```

Figure 4-3: No privatization due to flow dependences

4.1.1 Privatizing blocks

Given a constraint edge e_fc between two siblings m_s and m_t that are children of a node n, we can use block privatization to remove the edge if it is due to a storage-related data dependence e_d, where e_d goes between two nodes s and t that are in the subtrees below m_s and m_t, respectively (see Figure 4-4). We do not use privatization to remove such an edge if there is a path of constraint edges due to flow data dependences from m_s to m_t. In Figure 4-3, we do not remove the constraint edge due to the anti dependence edge from S_2 to S_4, since the constraint edges due to the flow dependence edges from S_2 to S_3 and from S_3 to S_4 cannot be removed. Also, we use block privatization to remove a constraint edge between two nodes m_s and m_t only if we can remove all of the constraint edges between m_s and m_t.

Since the constraint edge that we remove is due to a storage-related data dependence edge from s to t, t must define elements v_i of some variable v. We use block privatization to make the definitions (t, v_i) define elements of a private instance of v within the subtree beneath m_t.[1] We can create a private instance of v private within m_t's subtree only if the subtrees beneath m_s and m_t do not need to share storage for v, which could occur in two cases:

- Nodes in m_t's subtree could use the values of v_i from m_s's subtree: in Figure 4-5(a), S_6 could use the same value of `x` that S_1 uses (m_s is S_1 and m_t is S_2).

[1] An alternative would be to make the use or definition of v at s private, but we would need to have `copyin` instead of `copyout`.

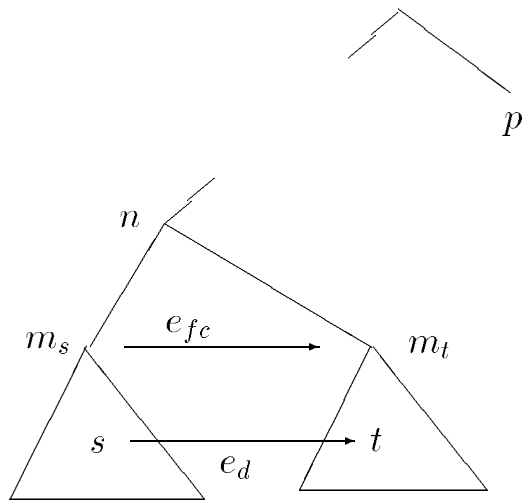

Figure 4-4: Forward control dependence graph during block privatization

```
(S1)   z = x - 4
(S2)   if p
(S3)       then begin
(S4)           if q
(S5)               then x = 19
(S6)           y = 40 * x
(S7)       end
```

Figure 4-5(a): Privatization fails due to flow of values

```
(S1)   z = x - 4
(S2)   if q
(S3)       then x = 19
(S4)   y = 40 * x
```

Figure 4-5(b): Privatization fails due to common use

Figure 4-5: No privatization

- The definition of the elements v_i in m_t's subtree could flow to a use of those elements (outside of the two subtrees) to which the value of those elements in m_s's subtree could also flow: in Figure 4-5(b), S_4 could use the value of `x` used in S_1 or the value of `x` defined in S_3 (m_s is S_1 and m_t is S_2).

If all of the elements v_i are dead at m_t, neither case is possible, and we can privatize. When we privatize `x` beneath m_t, nodes in a subtree beneath another sibling m_u may need m_t's private copy of `x`; if that is the case, m_u's subtree will be in the same parallel block as m_t and will correctly reference m_t's private copy, since there must be a flow constraint edge from m_t to m_u.

The immediate post-dominator p of n is the first statement executed after the forward control dependence subtree beneath n finishes executing; if a definition (t, v_i) reaches p and v_i is live at p, the values of elements of the private copy of v could be used later, and the variable must be copied out.

```
(S1)   if p
           then begin
               z = x - 4
               x = 19
               y = 40 * x
           end
(S7)   a = 30 * b
```

(S_1) `if p`
(S_2) `then begin`
(S_3) `z = x - 4`
(S_4) `x = 19`
(S_5) `y = 40 * x`
(S_6) `end`
(S_7) `a = 30 * b`

Figure 4-6(a): Simple privatization

(S_1) `if p`
(S_2) `then begin`
(S_3) `z = x - 4`
(S_4) `if q`
(S_5) `then begin`
(S_6) `x = 19`
(S_7) `w = 30 - x * 4`
(S_8) `end`
(S_9) `x = 23`
(S_{10}) `y = 40 * x`
(S_{11}) `end`
(S_{12}) `a = 30 * b`

Figure 4-6(b): More complex privatization

Figure 4-6: Privatization examples

In Figure 4-6(a), we would use privatization to remove the constraint edge from S_3 to S_4 (which is due to an anti-dependence between the same statements). Since the flow constraint edge from S_4 to S_5 ensures that S_4 and S_5 will be in the same parallel block, S_4 and S_5 will both reference the same private copy of `x`. The definition of `x` at S_4 reaches the immediate post-dominator S_7 of S_1; if `x` is live at S_7, we must copy out that private copy of `x`.

In Figure 4-6(b), we would privatize `x` at S_6 to remove the constraint edge from S_3 to S_4 (which is due to an anti-dependence edge from S_3 to S_6). This is possible since `x` is not live at S_4. S_6 and S_7 will reference the same private copy of `x`, since the flow constraint edge between the two nodes ensures that they will be in the same parallel block. We would also privatize `x` at S_9 to remove the constraint edges from S_3 and S_4 to S_9 (which are due to the anti dependence from S_3 to S_9, the output dependence from S_6 to S_9, and the anti dependence from S_7 to S_9); the flow constraint edge from S_9 to S_{10} will ensure that S_9 and S_{10} are in the same parallel block. Since the definition of `x` at S_6 cannot reach the immediate post-dominator S_{12} of S_1, that private copy of `x` does not need to be copied out. The definition of `x` at S_9 does reach S_{12}, and if `x` is live at S_{12}, that private copy of `x` must be copied out.

Block privatization can be performed upon both array and non-array variables. In Figure 4-7(a), the `i` loop could be run in parallel with the `j` and `k` loops, if the `j` and `k` loops had their own copies of the array `A`. The following data dependences exist in the program:

- a flow dependence from S_2 to S_3 over the elements of `A`.
- an output dependence from S_2 to S_6 over the odd elements of `A`.
- an anti dependence from S_3 to S_6 over the odd elements of `A`.
- an output dependence from S_2 to S_9 over the even elements of `A`.
- an anti dependence from S_3 to S_9 over the even elements of `A`.

```
(S1)    do i = 1 to 2*n
(S2)        A(i) = 3 * i - 44
(S3)        B(i) = 23 * A(i) - 8
(S4)    end do
(S5)    do j = 1 to n
(S6)        A(2j-1) = 13 * j - 2
(S7)    end do
(S8)    do k = 1 to n
(S9)        A(2k) = 13 * k - 2
(S10)   end do
```

Figure 4-7(a): Input program

```
        cobegin
            begin
(S1)            doall i = 1 to 2*n
(S2)                A(i) = 3 * i - 44
(S3)                B(i) = 23 * A(i) - 8
(S4)            end doall
            end
            begin
                private A copyout
(S5)            doall j = 1 to n
(S6)                A(2j-1) = 13 * j - 2
(S7)            end doall
            end
            begin
                private A copyout
(S8)            doall k = 1 to n
(S9)                A(2k) = 13 * k - 2
(S10)           end doall
            end
        coend
```

Figure 4-7(b): Output program

Figure 4-7: Privatization for arrays

We do not use privatization to remove the first data dependence, since it is a flow dependence. The elements of `A` over which the second and third data dependences exist are all dead at S_5, which implies that we can privatize `A` within the `j` loop. The elements of `A` over which the last two data dependences exist are all dead at S_8, which implies that we can privatize `A` within the `k` loop. Since S_6 and S_9 refer to different elements of `A`, there is no output dependence between them; the `j` and `k` loops can be run in parallel. Each parallel block only copies out the elements of `A` that it modifies; since there is no output dependence, neither block will copy out to the same storage location. The resulting output code is shown in Figure 4-7(b).

We represent block privatization by a *block privatization mark*, which is either $(n, m_t, v, \textbf{copyout})$ or $(n, m_t, v, \textbf{nocopyout})$. This mark specifies that the subtree beneath m_t receives a private copy of v.

Algorithm 1 To privatize blocks beneath a node n, call *privatize-block*(n).

Procedure *privatize-block*(n):

- We can remove a constraint edge $e_{fc} = ((m_s, m_t), n, e_d)$, if there is no path of flow constraint edges from m_s to m_t, where e_d is a storage-related data dependence edge from s to t over elements v_i of the variable v, and if all the elements v_i are dead at m_t (see Figure 4-4). Since e_d is a storage-related data dependence edge (an anti or output edge), (t, v_i) must be definitions in the program. Let p be the immediate post-dominator of n.
  - If any definition (t, v_i) reaches p and v_i is live at p, add a block privatization mark $(n, m_t, v, \textbf{copyout})$.
  - If the above condition is not satisfied, add a block privatization mark $(n, m_t, v, \textbf{nocopyout})$.

4.1.2 Privatizing loops

We can loop privatize a non-array variable v if each iteration uses a completely separate copy of v; loop privatization can remove some constraint marks due to loop-carried anti and output dependences, but it cannot remove those due to loop-carried flow dependences. We do not loop privatize array variables, since loop privatization is similar to scalar expansion. Also, we use loop privatization to remove a constraint mark at a node n only if we can remove all of the constraint marks at n.

If v is live at the header of a loop, an iteration may explicitly or implicitly use the value of v from a previous iteration, and we cannot privatize. In the program in Figure 4-8, we cannot privatize `x` in the `do` loop, since all of the iterations conditionally define `x`; the value of `x` after the loop ends could be the value defined in any iteration of the loop, since each iteration of the loop implicitly uses the previous iteration's value of `x`. However, if v is dead at the header of a loop, then each iteration defines its own value for v; each iteration can receive a private copy of v. We represent this via a *loop privatization mark*, which can be either $(n, v, \textbf{copyout})$ or $(n, v, \textbf{nocopyout})$. We test at the post-dominator for copyout as we did for block privatization.

Algorithm 2 To privatize a loop whose `do` node is n, call *privatize-loop*(n).

Procedure *privatize-loop*(n):

- We can remove a constraint mark (n_{do}, e_d) if e_d, a data dependence edge over the variable v, is not a flow data dependence edge, if v is a non-array variable, and if v is dead at the header node of the loop corresponding to n_{do}.

  - If v is live at the immediate post-dominator of n_{do}, add a loop privatization mark $(n, v, \textbf{copyout})$.
  - Otherwise, add a loop privatization mark $(n, v, \textbf{nocopyout})$.

```
(S1)   do i = 1 to n
(S2)       if p
(S3)           then x = 5 * i
(S4)   end do
(S5)   z = 5 * x - 4
```

Figure 4-8: No privatization

4.2 Explicit synchronization

To increase the amount of parallelism in our program, we may wish to introduce explicit synchronization (i.e. semaphores) into the output code. Constraint edges between two nodes can be satisfied in two ways: the sequentialization of their subtrees (implicit synchronization), or the insertion of explicit synchronization within the subtrees. In Figure 4-9, using implicit synchronization, we would execute the block of code from S_1 to S_4 before the block of code from S_5 to S_7. However, this does not give us maximal parallelism; the only statement in the second block that must wait for the first block is S_7. To use explicit synchronization, we would insert a `signal` statement after S_4 and a `wait` statement before S_7; we could then run the two blocks in parallel with each other.

We present an algorithm to insert explicit synchronization that can remove some constraint edges; however, more research needs to be done to decide when explicit synchronization should be used instead of implicit synchronization. If we use more explicit synchronization, we can achieve more parallelism, but the cost of using many semaphores is not negligible.

We add information to the f-control dependence graph to represent the addition of synchronization statements (`signal` and `wait` semaphore statements, and `semaphore` declarations). A semaphore whose scope is a `doall` loop is automatically private within the loop; this allows us to generate semaphores for loop-independent data dependences within

```
(S1)    x = 45
(S2)    y = 46
(S3)    z = x * y + 44
(S4)    w = 80 - z + 3 * x
(S5)    a = 91
(S6)    b = 83 - a
(S7)    c = w - b + 5 * a
```

Figure 4-9: Synchronization example

`doall` loops.[2] In Figure 4-10, each iteration of the `doall` loop contains its own copy of the semaphore `s1`, and none of the iterations will conflict in their use of semaphores.

We only generate explicit synchronization to remove constraint edges (constraints due to loop-independent data dependences); constraint marks are very difficult to remove, since synchronization for loop-carried dependences would require a statement in one iteration to refer to some previous iteration's semaphores. Also, our analysis does not give us enough information to add synchronization for loop-carried dependences efficiently: we do not know how many iterations a loop-carried data dependence crosses, only in what direction it crosses them.

We also do not add synchronization to remove all constraint edges. If a constraint edge is caused by a loop-independent data dependence edge from n to m, we only remove the constraint edge if n and m have the same immediately surrounding loop. In the program in Figure 4-11, it would be difficult to use semaphores to remove the constraint edge due to the loop-independent flow dependence from S_4 to S_7 over the elements of `A`, because we cannot be sure which iteration of the `j` loop assigns the final value to `A(i)`.

Given a loop-independent data dependence edge e_d from m to n, n should execute only after m executes. We ensure this by adding a `signal` s statement after m and a `wait` s statement after n. However, we cannot add a `signal` s statement only after m: if m does

[2] A semaphore whose scope is a `do` loop need not be private within each iteration, but the semaphore must be reinitialized by the run-time system every time a new iteration of the `do` loop begins.

```
(S1)   doall i = 1 to n
(S2)       private a copyout
(S3)       private b copyout
(S4)       private d copyout
(S5)       private e copyout
(S6)       semaphore s1
(S7)       cobegin
(S8)           if p
(S9)               then begin
(S10)                  a = 3 * i
(S11)                  signal s1
(S12)                  b = 5 * a - 4
(S13)              end
(S14)              else begin
(S15)                  a = 4 * i
(S16)                  signal s1
(S17)                  b = 6 * a - 25
(S18)              end
(S19)          if q
(S20)              then begin
(S21)                  d = 65
(S22)                  wait s1
(S23)                  e = d - a + 3
(S24)              end
(S25)              else d = 23
(S26)      coend
(S27)  end doall
```

Figure 4-10: Private semaphores

```
(S1)   do i = 1 to n
(S2)       do j = 1 to m
(S3)           if p
(S4)               then A(i) = 45 * i + j
(S5)       end do
(S6)       do k = 1 to l
(S7)               B(i,k) = 33 * A(i) - k
(S8)       end do
```

Figure 4-11: Non-removable constraint edge

not execute, n should not wait for it. Therefore, we need to add `signal` s statements at decision points that can avoid executing m; every node on the f-control dependence path from *Entry* to m is such a decision point (see Figure 4-12). The decision points on that path that are also ancestors of $LCA(m,n)$ do not require `signal` s statements, though: if the flow of control avoids $LCA(m,n)$, it will avoid n.[3] Since `signal` s and `wait` s statements need only be added in the subtree beneath $LCA(m,n)$, the scope of s can be limited to that subtree. We represent the scope of s via a *semaphore declaration mark*, which is of the form $(LCA(m,n), s, e_{lca})$, where $LCA(m,n)$ indirectly controls m and n via e_{lca} (see Lemma 2 on page 36).

Along the path p in the forward control dependence graph from $LCA(n,m)$ to m, we have to ensure that `signal` s is executed if the flow of control avoids m. Given a node p_i on p, where $p_i \neq LCA(m,n)$ and $p_i \neq m$, and the forward control dependence edge from p_i to p_{i+1} via e_f, m will not be executed if the control flow exit $e_t \neq e_f$ is taken from p_i. We represent this via a *path signal mark*, which is of the form (p_i, s, e_t). We do not need to add path signal marks at $LCA(m,n)$, since if e_{lca} is not taken from $LCA(m,n)$, neither m nor n will execute.

If m is not a decision point in the program, we can insert a `signal` s statement

[3] The same information could be computed by using data flow analysis instead of control dependence [CKM88].

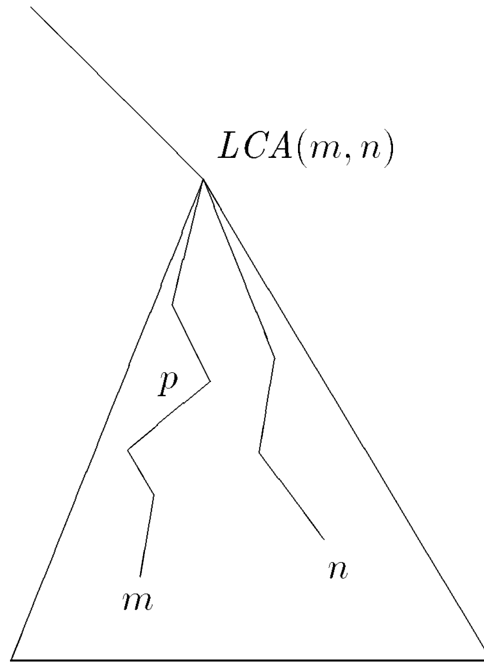

Figure 4-12: Adding synchronization to the forward control dependence graph

immediately after m; we represent this via a *node signal mark*, which is of the form (m, s). Otherwise, if m is a decision point, we must execute `signal` s on both control flow exits, e_t and e_f, from m; we use path signal marks (m, s, e_t) and (m, s, e_f) to represent this.

We can simply insert a `wait` s statement immediately before n; we represent this via a *wait mark*, which is of the form (n, s).

Algorithm 3 We can remove a constraint edge due to a loop-independent data dependence $e_d = ((m, n), v, t, \textbf{independent})$ by calling *synch*(e_d), if the same loop immediately surrounds both n and m.

Procedure *synch*(e_d):

- Create a new semaphore s whose scope is the subtree of statements beneath $LCA(m, n)$; add a semaphore declaration mark $(LCA(m, n), s, e_{lca})$, where $LCA(m, n)$ indirectly controls m and n via e_{lca}.

- Do the following for each node p_i, $p_i \neq LCA(m, n)$ and $p_i \neq m$, that lies on the path p from $LCA(m, n)$ to m in $G_{fc}(P)$:

  - Let $e_p = ((p_i, p_{i+1}), e_f) \in E_{fc}(P)$ be the f-control dependence edge with tail p_i that is on the path p.

  - We must execute `signal` s when e_t is taken, where $e_t \in E_f(P)$ has tail p_i, and $e_t \neq e_f$. Add a path signal mark (p_i, s, e_t).

- We must execute `signal` s immediately after m executes. If m has children in $G_{fc}(P)$, insert path signal marks (m, s, e_t) and (m, s, e_f), where $e_t, e_f \in E_f(P)$ are the control flow arcs that have tail m. If m has no children in $G_{fc}(P)$, add a node signal mark, which is of the form (m, s).

- We must execute `wait` s immediately before n executes; add a wait mark (n, s).

4.3 Code generation

We generate code during a single pass over the constrained forward control dependence graph. The pass is handled by two mutually recursive procedures: one procedure generates siblings beneath a node, and the other generates the code for a node itself. We begin by generating code for the forward control dependence children of *Entry*.

When we generate code for the children of a node n that are controlled via an edge e_f, we partition the children into constraint-edge-connected sets: each such set can be run in parallel with other such sets. Within each set, we can generate sequential code for the nodes in any order consistent with that specified by the constraint edges. Our algorithm first generates code for all of the children that have no incoming constraint edges. We then generate all of the children who only have incoming constraint edges from already generated siblings; we repeat this step until we have generated all of n's children.

Algorithm 4 Call *child-gen*(*Entry*,e_{start}), where e_{start} is the control flow edge in $G_f(P)$ with tail *Entry*.

Procedure *child-gen*(n, e_f):

- For every semaphore declaration mark (n, s, e_f), output a `semaphore` s statement.

- For every path signal mark (n, s, e_f), output a statement `signal` s.

- Output a `cobegin` statement.

- Repeat the following for each constraint-edge-connected set $S = \{s_i\}$ of nodes that are controlled by n via e_f.

  - Output a `begin` statement.

  - Repeat the following for each node $s_i \in S$:

    - Output a statement `private` v_i for each block privatization mark $(n, s_i, v_i, \textbf{nocopyout})$.

    - Output a statement `private` v_i `copyout` for each block privatization mark $(n, s_i, v_i, \textbf{copyout})$.
  - Let $d(s_i)$ be the maximum length of a path of constraint edges from $s_i \in S$ to a node in S with no incoming constraint edges. Repeat the following, with d_{cur} going from 0 to $max(d(s_i))$.
    - Repeat the following for each node $s_i \in S$ such that $d(s_i) = d_{cur}$:
      - Call *code-gen*(s_i).
  - Output an `end` statement.
- Output a `coend` statement.

Procedure *code-gen*(n):

- If n is an unconditional `goto`, `return`, `begin`, `end` or `end do` statement, do nothing.
- If there exists a wait mark (n, s), output a `wait` s statement.
- If n is a `do` node:
  - If there exists a constraint mark (n, dd), output a `do` statement.
  - If there do not exist any constraint marks (n, dd), do the following:
    - Output a corresponding `doall` statement.
    - Output a statement `private` v_i for every loop privatization mark $(n, v_i, \textbf{nocopyout})$.
    - Output a statement `private` v_i `copyout` for every loop privatization mark $(n, v_i, \textbf{copyout})$.
  - Call *child-gen*(n, e_t), where $e_t = (n, h)$, and h is the header of n's loop; this generates code for the body of the loop.

- Output a corresponding `end do` or `end doall` statement.

- If n is an `if-then-else` node:
  - Output code for n.
  - Output `then`, and call *child-gen*(n, e_t), where e_t is the control flow edge taken under the *then* exit from n; this generates the code for the *then* clause of the conditional.
  - Output `else`, and call *child-gen*(n, e_f), where e_f is the control flow edge taken under the *else* exit from n; this generates code for the *else* clause of the conditional.

- If n has no children in $G_{fc}(P)$:
  - Output code for n.
  - If there exists a node signal mark (n, s), output a `signal` s statement.

This algorithm could generate extra `begin/end` scopes in certain places, such as around single statements within a `cobegin` block. These could be eliminated later by performing a "clean-up" pass over the generated program.

Additional parallelism could possibly be generated within each constraint-edge-connected set of children; those siblings that do not have a path of constraint edges connecting them could potentially be run in parallel. However, this is an area for further research: problems arise in connection with privatization, because we must ensure that statements still reference the correct storage locations.

Chapter 5

Conclusions

The constrained forward control dependence graph allows us to represent more parallelism than that allowed by current parallelizers; we are able to generate nested parallelism, with parallel loops and parallel blocks. However, this could produce too much fine-grained parallelism, which could result in the output program running more slowly than the input program! Research must be done in the area of cost analysis, so that a compiler can correctly decide how much parallelism to generate. Tradeoffs between several factors are involved — the amount of parallelism, the amount of explicit synchronization, and the amount of extra memory due to privatization.

The tradeoff between parallelization and vectorization must also be examined. On machines with multiple CPUs and multiple vector or array processors, inner `do` loops should probably be vectorized rather than parallelized, but it is not clear what combination of vectorization and parallelization would be optimal.

The generation of more general privatization and synchronization is another area for further research. We should also be able to perform loop privatization upon arrays, as well as generate synchronization for all data dependences, not just some loop-independent data dependences.

An alternative to the `copyout` statement would be a `copyin` statement; we believe that both statements have equal expressive power, but perhaps one is more "natural"

to use. Another issue that needs to be examined is whether parallelizers can make use of non-determinism within parallel languages, since non-deterministic parallel constructs are useful in expressing many parallel algorithms.

Methods for computing data flow and data dependence information for individual elements of arrays need to be improved; the more precise such information is, the more parallelism we can generate.

We have implemented our program representation, constrained forward control dependence, in the PTRAN system at IBM Research, along with the privatization and code generation algorithms; the synchronization algorithm is not implemented. Our implementation of constrained forward control dependence is more general that described in this thesis, in that it handles most non-structured, reducible code.

We have run PTRAN on EISPACK [SBDG76] and LINPACK [DBMS79], and the results look promising. However, we do not have any data available for this thesis; actual measurements will be published later [BCFH88].

Appendix A

Computing constraints

Algorithm 5 This algorithm computes constraint edges. For every loop-independent data dependence edge $e_d = (n, m)$ in $E_d(P)$, call *markup-block*(e_d).

Procedure *markup-block*(e_d):

- If $LCA(n, m) = n$ or $LCA(n, m) = m$, then do nothing.

  Otherwise, by Lemma 2, there must exist two control dependence edges $e_n = ((LCA(n, m), n_0), e_f)$ and $e_m = ((LCA(n, m), m_0), e_f)$ in $E_{fc}(P)$ such that n_0 is an ancestor of n in $G_{fc}(P)$, and m_0 is an ancestor of m in $G_{fc}(P)$. Add the constraint edge $((n_0, m_0), LCA(n, m), e_d)$.

Algorithm 6 This algorithm computes constraint marks. For every loop-carried data dependence edge e_d in $E_d(P)$, call *markup-loop*(e_d).

Procedure *markup-loop*(e_d):

- Let n_{do} be the `do` node of the loop that carries the data dependence; add the constraint mark (n_{do}, e_d).

Bibliography

[ABCC87] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the ptran analysis system for multiprocessing. In *Proceedings of the 1987 International Conference on Supercomputing*, Springer-Verlag, 1987. To appear in a special issue of the Journal of Parallel and Distributed Computing.

[AK87] J.R. Allen and K. Kennedy. Automatic transformation of fortran programs to vector form. In *ACM Transactions on Programming Languages and Systems*, pages 491–592, October 1987.

[All83] John Randal Allen. *Dependence Analysis for Subscript Variables and its Application to Program Transformation.* Ph.D. dissertation, Rice University, 1983.

[AS83] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *Computing Surveys*, 15(1):3–43, March 1983.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[Ban79] Utpal Banerjee. *Speedup of Ordinary Programs.* Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1979.

[BCFH87] Michael Burke, Ron Cytron, Jeanne Ferrante, and Wilson Hsieh. *Automatic Determination of Private and Shared Variables for Nested Processess.* Technical Report RC 13194, IBM T.J. Watson Research Center, August 1987.

[BCFH88] Michael Burke, Ron Cytron, Jeanne Ferrante, Wilson Hsieh, David Shields, and Vivek Sarkar. *On the Automatic Generation of Useful Parallelism: A Tool and an Experiment (Extended Abstract).* Technical Report, IBM T.J. Watson Research Center, January 1988. Accepted for ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems.

[BJ66] C. Böhm and G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. In *Communications of the ACM*, pages 366–71, Association for Computing Machinery, May 1966.

[Bur87] Michael Burke. *An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data Flow Analysis.* Technical Report RC 12702, IBM T.J. Watson Research Center, Sept 1987.

[CF87a] Ron Cytron and Jeanne Ferrante. *An Improved Control Dependence Algorithm.* Technical Report RC 13291, IBM T.J. Watson Research Center, 1987.

[CF87b] Ron Cytron and Jeanne Ferrante. What's in a name? -or- the value of renaming for parallelism detection and storage allocation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, St. Charles, IL, August 1987.

[CKM88] Ron Cytron, Steve Karlovsky, and Kevin P. McAuliffe. *Automatic Management of Programmable Caches.* Technical Report CSRD Report No. 728, University of Illinois Center for Supercomputing Research and Development, January 1988. Submitted to 1988 International Conference on Parallel Processing, St. Charles, IL.

[Cyt87] Ron Cytron. Lecture notes. June 1987. Lecture notes for course taught at NYU by Ron Cytron.

[DBMS79] J.J. Dongarra, J.R. Bunch, C.B. Moler, and G.W. Stewart. *Linpack Users' Guide*. SIAM Press, Philadelphia, Pennsylvania, 1979.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. In *ACM Transactions on Programming Languages and Systems*, pages 319–349, Association for Computing Machinery, July 1987.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs*. North-Holland, New York, NY, 1977. Out of print.

[IBM88] IBM. *Parallel Fortran Language and Library Reference*. International Business Machines, March 1988.

[KKLW80] D.J. Kuck, R.H. Kuhn, B. Leasure, and M. Wolfe. The structure of an advanced vectorizer for pipelined processors. In *Proceedings of CompSAC80 (Fourth International Computer Software and Applications Conference)*, pages 709–715, October 1980.

[Kuc78] David J. Kuck. *The Structure of Computers and Computation*. Volume 1, John Wiley and Sons, New York, NY, 1978.

[Lea85] Bruce Leasure. *The Parafrase Project's Fortran Analyzer, Major Module Documentation*. Technical Report CSRD Report No. 504 PR-85-5, UILU-ENG-85-8005, University of Illinois at Urbana-Champaign Center for Supercomputing Research and Development, July 1985.

[LM77] R.C. Linger and H.D. Mills. On the development of large reliable programs. In Raymond T. Yeh, editor, *Current Trends in Programming Methodology, Vol-*

ume I: Software Specification and Design, chapter 5, pages 120–139, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.

[Mil82] H.D. Mills. Mathematical foundations for structured programming. In Edward Yourdon, editor, *Writings of the Revolution: Selected Readings on Software Engineering*, chapter 14, pages 220–226, YOURDON Press, New York, NY, 1982.

[PW86] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[SBDG76] B.T. Smith, J.M. Boyle, J.J. Dongarra, B.S. Garbow, Y. Ikebe, V.C. Klema, and C.B. Moler. *Matrix Eigensystem Routines - Eispack Guide.* Springer-Verlag, 1976.

[Wol82] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1982.

On page 52 of the thesis, the specification of **Algorithm 1** contains an error. The procedure ***privatize-block*** should contain the additional condition that v can be privatized only if one of following is true:

- all of the definitions (t, v_i) reach p.
- none of the definitions (t, v_i) reach p.
- none of the elements v_i is live at p.