Resource-Constrained Software Pipelining

Alexander Aiken
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720-1776
email: aiken@cs.berkeley.edu

Alexandru Nicolau[*]
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
email: nicolau@ics.uci.edu

Steven Novack[†]
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
email: snovack@ics.uci.edu

Abstract

This paper presents a software pipelining algorithm for the automatic extraction of fine-grain parallelism in general loops. The algorithm accounts for machine resource constraints in a way that smoothly integrates the management of resource constraints with software pipelining. Furthermore, generality in the software pipelining algorithm is not sacrificed to handle resource constraints, and scheduling choices are made with truly global information. Proofs of correctness and the results of experiments with an implementation are also presented.

1 Introduction

Recently there has been considerable interest in a class of compiler parallelization techniques known collectively as *software pipelining.* Software pipelining algorithms compute a static parallel schedule overlapping the operations of a loop body in much the same way that a hardware pipeline overlaps operations in a dynamic instruction stream. The schedule computed by a software pipelining algorithm is suitable for execution on a synchronous, tightly-coupled parallel machine, such as a super-scalar or VLIW (Very Long Instruction Word) machine.

Software pipelining algorithms are interesting for at least three reasons. The first reason is that super-scalar and VLIW machines are being built. IBM's System 6000 can execute four operations in parallel; Intel's i860 and i960 chips can execute three operations in parallel. The largest tightly-coupled synchronous machine built to date is Multiflow's TRACE-14, which has 14 functional units. Several computer manufacturers—e.g., HP, Phillips, Siemens—are also developing VLIW or super-scalar architectures. The second reason is that these tightly-coupled machines must be programmed at a very low

[*]This work supported in part by ONR grant # N0001490K0215
[†]This work supported in part by ONR grant # N0001490K0215

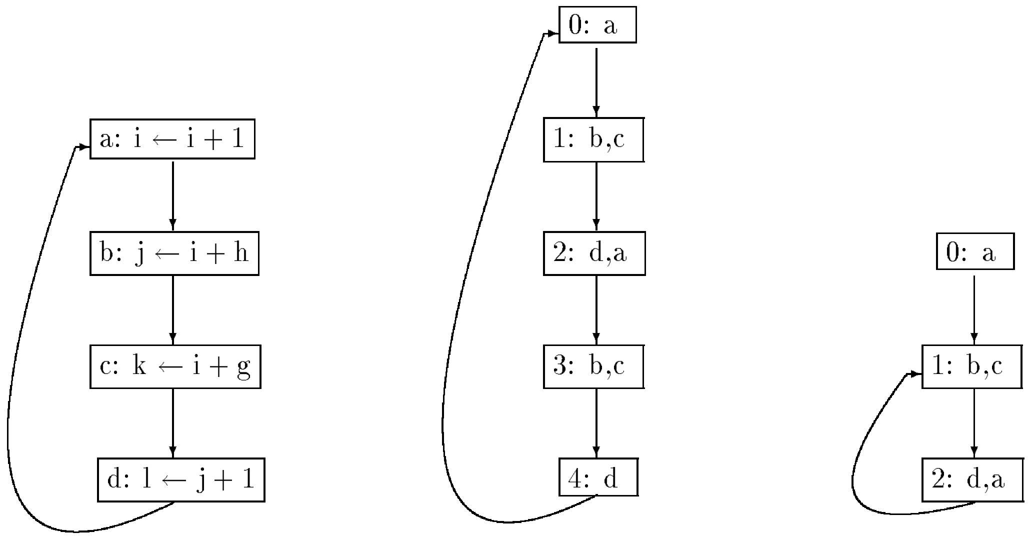
level. Someone writing a program for a tightly-coupled machine must develop a parallel schedule, which means that person must know about and account for details of the hardware design such as instruction timings and resource conflicts between functional units. This task is extremely time-consuming and error-prone; compilation techniques are needed to translate programs written at a reasonably high level into good parallel schedules.

The final reason is that software pipelining techniques hold the promise of producing better code with faster compilation time than other scheduling techniques. This potential is illustrated by the example in Figure 1. Figure 1a shows a simple sequential loop and Figures 1b and 1c show two different parallel schedules for the loop. For convenience, we label the operations in the original loop a–d and refer only to these labels in the parallel loops. In this example, some parallelism is present within the loop body (because operations b and c can be executed simultaneously) as well as across iterations (because d from one iteration can overlap with a from the next iteration). The classical approach to scheduling the loop in Figure 1a is to unroll the loop body some number of times and then apply scheduling heuristics within the unrolled loop body [Fis81] as illustrated in Figure 1b. While this approach allows parallelism to be exploited between some iterations of the original loop, there is still sequentiality imposed between iterations of the unrolled loop body. In general, if the loop could be fully unrolled, all parallelism both inside and across iterations could be exploited by this approach. However, full unrolling is usually impossible or impractical to obtain. Software pipelining provides a direct way of exploiting parallelism inside and across all iterations of a loop; hence software pipelining achieves the effect of scheduling with full unrolling. A software-pipelined version of the original loop is given in Figure 1c.

1.1 Previous Work

One body of work on software pipelining has focussed on establishing the formalism required to adequately address what software pipelining algorithms can and cannot achieve. Results in this line of development include a software pipelining algorithm that generates optimal code for loops without conditional tests [AN88a] and a proof that optimal software pipelining is impossible in general [SGE91]. However, this work has largely ignored resource constraints.

Existing software pipelining algorithms handle resource constraints in a variety of ways. Some algorithms deal with only weak forms of resource constraints—e.g., the number of operations that can be executed in parallel. Others assume resource constraints are handled in a separate "fix-up phase" after software pipelining [NPA88]. Several software pipelining algorithms account for resource constraints directly as part of the software pipelining algorithm, e.g. [RG81, Lam87]. However, in most such algorithms the treatment of resource constraints is intimately connected to software pipelining—that is, the software pipelining is not separable from the handling of resource constraints. One of our interests is to separate what is really intrinsic to software pipelining from other, orthogonal concerns. A more extensive discussion of previous and related work is included in Section 9.



(a) An example loop. (b) Unrolled twice and scheduled. (c) Pipelined loop.

Figure 1: Loop Unrolling and Software Pipelining

1.2 Our Approach

In this paper, we present an algorithm that smoothly integrates software pipelining with the treatment of resource constraints, while at the same time maintaining a structured design that separates orthogonal concerns. Our algorithm serves two purposes. First, we believe the algorithm represents a practical direction and can form the basis of implementations of software pipelining; we discuss an implementation of our algorithm in Section 7. Second, the algorithm represents a summary of many of the most interesting aspects of our investigation of software pipelining over the last several years [Nic85, AN88a, AN88b, Aik88, Aik90, AN91]. Our algorithm has several novel features:

- The handling of resource constraints is orthogonal to the software pipelining.
- At each step the algorithm has global information about the operations that can be scheduled.
- In a technical sense defined precisely in Section 8, given sufficient resources our algorithm can produce code arbitrarily close to the theoretical optimum.

The advantage of the first point is that the treatment of resources could be modified (say, for a different machine) and no changes would be required in the overall algorithm. The second and third points together imply that the quality of the final pipelined loop is limited only by the ability to make good resource allocation decisions (see Section 8) and not by the design of the software pipelining algorithm.

Our software pipelining algorithm is built from two components: a *scheduler* and a *dependence analyzer*. The machine-dependent *scheduler* is used to incrementally build a parallelized loop from a sequential loop. For each parallel instruction, the scheduler selects operations to schedule based on the set of operations available for scheduling in that instruction and available resources. The set of *available operations* is maintained by a global dependence analyzer; as the scheduler makes decisions about where to place operations, the set of available operations is updated incrementally. Together, the scheduler and the dependence analyzer encapsulate all machine-dependent information. As the parallelized loop is constructed, the software pipelining algorithm checks for repeating states that can be "pipelined." The software pipelining algorithm itself is very simple; the difficulty lies in establishing minimal restrictions on the scheduler and dependence analyzer that guarantee the correctness and termination of the software pipelining algorithm.

The rest of this paper is divided into nine sections. Section 2 defines the model of parallel computation used to develop the algorithm. Section 3 works through a small example to give an intuitive idea of how the software pipelining algorithm works. Section 4 describes the algorithm and presents a proof of correctness. Section 5 gives an algorithm for incrementally maintaining the set of available operations. Section 6 describes the integration of resource constraints into the algorithm. Handling resources well is critical in realistic applications of software pipelining. Section 7 briefly describes an implementation of our algorithm, some additional optimizations, and some experimental results. The experimental results bear out the strengths of our approach and point out some weaknesses; both are discussed at length. Section 8 presents a result that suggests our algorithm can achieve the best schedules possible in the presence of resource constraints. A discussion of related work is in Section 9. The final section summarizes and presents some conclusions.

2 Basic Terminology

This section develops a simple model of a tightly-coupled, synchronous parallel machine. The formalism is used to explain our software pipelining algorithm and to provide a basis for a proof of correctness.

A *program* is an automaton $\langle \mathcal{X}, \delta, n_0, \mathcal{N} \rangle$. $\mathcal{X}$ is a set of n operations $\{x_0, \ldots, x_{n-1}\}$. Operations are divided into *assignments* that read and write a global store, *tests* (boolean-valued functions) that affect the flow of control, and a distinguished operation *stop*.

The body of the program is a set $\mathcal{N}$ of *states* $n_0, \ldots, n_{m-1}$. The state n_0 is the *start state* of the program. Associated with each state n is $ops(n)$, the operations of n, which are elements of $\mathcal{X}$. The states represent parallel instructions; intuitively, when control reaches a state n, all operations in $ops(n)$ are executed simultaneously. To simplify the presentation, we assume that all operations execute in unit time. Extensions to multi-cycle operations and pipelined functional units are discussed in Section 6.

A *configuration* is a pair $\langle n, s \rangle$ where n is a state and s is a store (the contents of memory locations and registers). The *transition function* δ maps configurations into configurations. An *execution* is a sequence of configurations $\langle \ldots, \langle n_i, s_i \rangle, \ldots \rangle$ such that $\delta(\langle n_i, s_i \rangle) = \langle n_{i+1}, s_{i+1} \rangle$.

The transition function describes how a tightly-coupled, synchronous machine actually executes a parallel instruction. We deliberately avoid defining a transition function in any detail. The transition functions of super-scalar and VLIW machines are complex and vary considerably from machine to machine. The greatest source of complexity is defining what it means to execute more than one test in parallel (multi-way jumps). As an example, in one possible model tests within a state n are always organized as a binary decision tree with a unique root. One branch of each test in the decision tree is labeled *true*, the other is labeled *false*. Each leaf of the decision tree is a pointer to another state. When the state is executed, all of the tests are evaluated in parallel in the store. The next state to be executed is the leaf that terminates the (unique) path from the root where every branch is labeled by the value of that test in the store. There are other possible implementations of multi-way jumps; many mechanisms have been proposed and implemented [Fis80, KN85, AAG$^+$86, Ebc87]. The software pipelining algorithm we present applies to any of these control-flow mechanisms.

We use the following abstraction of control-flow throughout this paper. We assume that control-flow is determined entirely by tests; that is, the result of evaluating the tests in a state determines the next state. A *branch* of a state n is a truth assignment $\langle x_1 = \mathit{true}, x_2 = \mathit{false}, \ldots \rangle$ to the tests $x_1, x_2, \ldots$ in n. The set of all branches of n is $branch(n)$; if n has no tests, then $branch(n)$ is the singleton set $\{\langle\rangle\}$ consisting of the empty truth assignment. The function *succ-on-branch* maps a state n and a branch $c \in branch(n)$ to a successor node n' (the name *succ-on-branch* stands for "successor on branch").[1] We

[1]Note that in most cases a node with k tests will not have 2^k distinct successors. For generality, we treat each of the 2^k branches separately in our algorithm; in an implementation for a particular control-flow mechanism many branches can be

assume that if *succ-on-branch*$(n, c) = n'$, then there is a state s such that $\delta(\langle n, s\rangle) = \langle n', s'\rangle$ and that the evaluation of tests in configuration $\langle n, s\rangle$ satisfies the truth assignment c.

The set of *successors* $succ(n)$ of a state n is $\{n' | \exists c\ s.t.\ \textit{succ-on-branch}(n, c) = n'\}$. When n is executed, control is transferred to some $n' \in succ(n)$. A state that contains the operation *stop* cannot contain other operations and cannot have any successors.

We next define a meaning function μ for programs, which is used in the proof that our software pipelining algorithm is correct (i.e. that it preserves the meaning of the original program).

Definition 2.1 Let P be a program $\langle \mathcal{X}, \delta, n_0, \mathcal{N}\rangle$. If there is an execution $\langle\langle n_0, s\rangle, \ldots, \langle n_k, s'\rangle\rangle$ such that $ops(n_k) = \{stop\}$ then $\mu(P, s) = s'$. If no such execution exists, then $\mu(P, s) = \perp$.

Programs P_1 and P_2 are *equivalent* ($P_1 \equiv P_2$) if $\forall s\ \mu(P_1, s) = \mu(P_2, s)$.

Software pipelining is a loop parallelization technique, so we must describe the loops we are interested in parallelizing. For convenience, we use the following definition. A *sequential loop* is a program with i operations $x_0, \ldots, x_{i-1}$ and i states $n_0, \ldots, n_{i-1}$ where $n_j = \{x_j\}$. All backedges go to the start state n_0; that is, if $n_i \in succ(n_j)$ and $i \leq j$ then $i = 0$. Every state is assumed to be reachable from the start state.

3 An Example

Given a sequential loop L, our software pipelining algorithm incrementally builds a parallelized loop from L. Initially, the parallelized loop is empty (has no states) and the algorithm chooses a set of operations from the sequential loop L that legally can be scheduled as the start state of the parallel loop. After scheduling a subset of the available operations as the start state, the algorithm recursively schedules the successors of the start state by considering what operations can be scheduled in the successor states, and so on. The main difficulty is guaranteeing that this procedure terminates. We show that eventually the scheduled states must fall into a detectable repeating pattern, at which point a loop can be constructed from this pattern of repeating states.

An important data structure used by the algorithm is an incrementally maintained set A of *available operations*. At each step, A contains a set of operations available for scheduling in the current state being constructed. How this set is built and maintained is discussed in Section 5. For now it is only important to understand that set A contains all operations that could be scheduled legally in the current state without violating program semantics.

Initially, the new program graph is empty and A contains all operations available for scheduling in the first state. Consider the program in Figure 2. We display programs as control-flow graphs with the convention that true branches of tests are to the left and false branches are to the right. Not all operations can be scheduled in the first state; for example, c must be scheduled after b, since c references a value that b writes. In standard compiler terminology, there is a *data dependence* from b to c [KKP+81].

For this example, we assume a machine model in which all reads take place before any writes during execution of a state, and write conflicts are not permitted. In this model, the operations a, b, and f are

treated together.

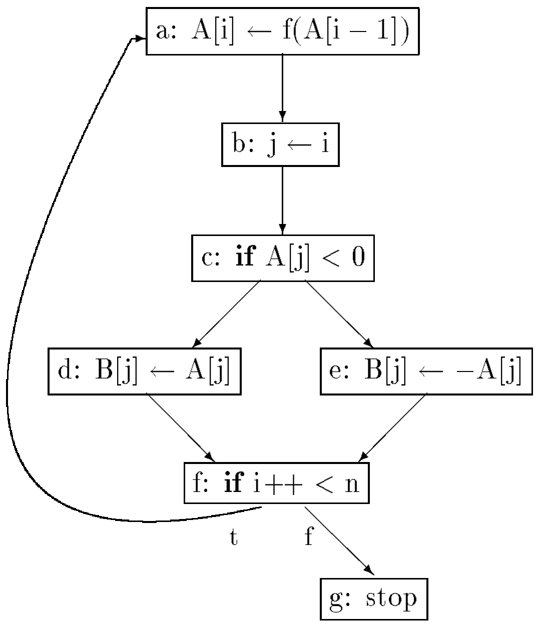


Figure 2: An example loop.

all available for scheduling in the first state. Because the algorithm may overlap operations from different iterations, we superscript operations with the scheduled iteration from which they came. In addition, we subscript available operation sets to keep track of different values for different states. Thus, initially $A_0 = \{a^0, b^0, f^0\}$.

Another component of the pipelining algorithm is the *scheduler.* The scheduler selects from A a set of operations to schedule in the current state. Together, the procedure to maintain the set of available operations and the scheduler encapsulate all machine-dependent information. The software pipelining algorithm itself is built on top of these two components.

A pipelined version of the loop in Figure 2 is given in Figure 3. In Figure 3, the state n_i is labeled by the integer i. The rest of this section describes how the software pipelining algorithm computes this parallel schedule from the sequential loop. For the first state n_0, assuming that the machine has sufficient resources, the scheduler could choose to schedule all available operations. Because f is a test, there will be two successors of the first state—one for the case where f evaluates to *true*, and one for the case where f evaluates to *false*. The sets of available operations are different for the two successors.

Consider the successor n_1 of n_0 for the case where f^0 evaluates to false. This case is easy, as the program terminates on this branch. The new set A_1 of available operations is $\{c^0, d^0, e^0\}$, reflecting the fact that a^0, b^0 and f^0 have been scheduled and that this branch of f^0 is the loop exit. Because write conflicts are not permitted, d^0 and e^0 cannot be scheduled in the same state, but both are "available"—at

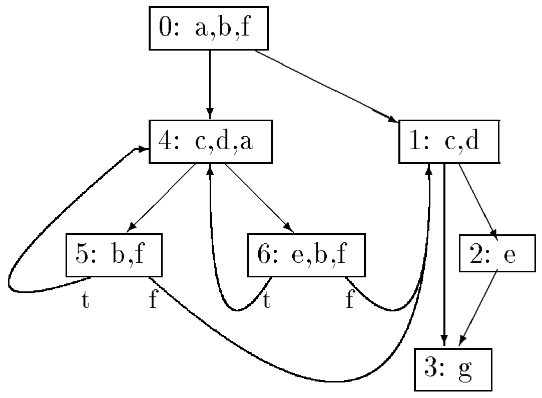
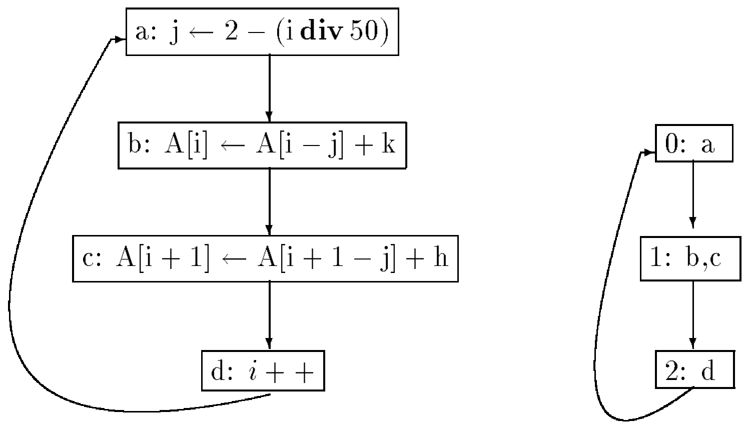


Figure 3: The loop after software pipelining.

this point, all dependences on the two statements have been satisfied. Assume that the scheduler selects operations c^0 and d^0 for state n_1. Operation c^0 is a test, so there are two successors of this state. For the successor n_2 where c^0 evaluates to false, the set of available operations A_2 is $\{e^0\}$. Assume that the scheduler places e^0 in n_2. For the single successor n_3 of n_2 the set of available operations A_3 is just $\{g^0\}$, the stop operation. Thus n_3 contains only g^0. Backing up to n_1, the set of available operations for the branch where c^0 evaluates to true is also $\{g^0\}$, so the successor of n_1 on this path is also n_3.

This completes the terminating path from n_0. On the other path, where f^0 evaluates to true, the new set of available operations A_4 is $\{c^0, d^0, e^0, a^1\}$. Note that the operation a^1 from the second iteration is available for scheduling in parallel with statements from the first iteration. A subtle point is that operation b^1 is not available for scheduling, even though all reads take place before all writes and all operations from the first iteration that read variable j are available in A_4. Operation b^1 is not available because (as before) d^0 and e^0 cannot be scheduled in the same state. Even though all operations that read j are in A_4, not all of these can be scheduled in n_4, and this fact prevents statements that write j from being available.

Assume that the scheduler selects operations c^0, d^0, and a^1 for state n_4. Operation c^0 is a test, so there are two successors of this state. For the successor n_5 where c^0 evaluates to true, the set A_5 is $\{b^1, f^1\}$. Assuming that the scheduler places both operations in n_2, the set of available operations for the successor of n_5 on the path where f^1 is true is $\{c^1, d^1, e^1, a^2\}$. Note that, except for the superscripts, this set is exactly the same as A_4. The superscripts are just a way of keeping track of the iteration of each operation; the sets have the same operations. Rather than continue scheduling at this point, the pipelining algorithm simply makes n_4 a successor of n_5. Similarly, the set of available operations for the successor of n_5 where f^1 evaluates to false is $\{c^1, d^1, e^1\}$. Except for superscripts, this is exactly the same as A_1. As before, the pipelining algorithm makes n_1 a successor of n_5.



(a) Another example loop. (b) An incorrect schedule.

Figure 4: Another example loop.

Backing up, the pipelining algorithm next considers the successor n_6 of n_4 where c^0 evaluates to false. The set of available operations A_6 is $\{e^0, b^1, f^1\}$. Assuming that the scheduler places all three operations in n_6, the sets of available operations for the two successors of n_6 are the same as for n_5 and scheduling proceeds just as it did for n_5. The algorithm terminates with the schedule in Figure 3.

There are three technically difficult aspects of the software pipelining algorithm. The first problem is justifying the step where previously scheduled states are "reused", such as when the pipelining algorithm decided to make n_4 the successor of n_5. We have simply implied that this is correct, and in the example it happens to be correct, but in general this step is not correct. Intuitively, the problem is that just because two sets of available operations happen to be the same for two different states, that does not by itself guarantee that all subsequent sets of available operations would be the same in all successors of those states.

We illustrate this problem with the loop in Figure 4a. To make the example as simple as possible, there are no conditional statements or exits from the loop. Assuming that the variable i is always zero upon entering the loop, note that statements b and c are independent for the first 50 iterations and data dependent for the next 50 iterations. If dependence analysis recognizes that b and c are independent for the first 50 iterations, then as the parallelized loop is built the scheduler could place b and c together in the first 50 iterations. Following the pipelining strategy for the previous example, repeating states would be detected in the second iteration, leading to the parallelized program in Figure 4b, which is clearly incorrect. In this example, irregular dependencies make it difficult to detect repeating behavior. Section 4 formalizes the software pipelining algorithm and provides constraints on the scheduler and available operation information that guarantee the correctness and termination of the software pipelining algorithm.

The second problem is computing the sets of available operations. An algorithm for maintaining these sets incrementally was first presented in [EN89] for programs without loops (i.e. with acyclic control-flow graphs). In Section 5, we present a detailed description of the computation and maintenance of available operations for use in software pipelining of loops. Our presentation is simpler and easier to understand and implement than the algorithm in [EN89].

The third significant problem is managing finite resources. While resource allocation does not bear directly on the correctness of our software pipelining algorithm, good resource usage is obviously important if the algorithm is to be useful in practice. In Section 6 we show how finite resources is integrated with software pipelining in our system.

4 The Software Pipelining Algorithm

The example in Section 3 illustrates that the key step in our algorithm is discovering when states can be "reused" to form a software pipeline. Recognizing patterns in the scheduled operations is not trivial and is in fact not valid if the scheduler and the available operation analysis are not constrained in some way. For example, if the scheduler merely selects operations to schedule at random, no repeating behavior can be inferred. Similarly, even if the scheduler is well-behaved, the example in Figure 4 shows that if the available operation analysis does not exhibit a detectable pattern, software pipelining is not possible.

In this section we present constraints on the scheduler and available operation analysis that make software pipelining possible. These constraints are quite weak and are easily satisfied in practice. After presenting the constraints, we present the software pipelining algorithm itself and prove its correctness. Finally, we discuss termination of the software pipelining algorithm.

4.1 The Constraints

Recall that x_i^c denotes the instance of operation x_i from iteration c of a loop. The following definition is used in the discussion of the constraints.

Definition 4.1 Let $X = \{\ldots, x_i^{j_i}, \ldots\}$ be a set of operations. The set X^c is the set $\{\ldots, x_i^{j_i+c}, \ldots\}$.

As discussed in Section 3, one component of the software pipelining algorithm is a scheduler for a specific machine. The following constraint requires that: (1) the scheduler is a function, (2) the scheduler must schedule some operation in every state, and (3) the operation chosen can depend on the set of operations available and the relative distance in iterations between the operations available, but not on the actual iterations of the operations available.

Constraint 4.2 Let X be a set of operations. The scheduler must be a function mapping a set of already scheduled operations and a set of available operations to a single operation or the value "none." In addition, $\mathit{schedule}(X, A) \neq$ none if $X = \emptyset$. We also require that

$$(\exists x_j\ \forall i\ \mathit{schedule}(X^i, A^i) = x_j^{k+i} \text{ and } x_j^{k+i} \in A^i) \ \bigvee\ (\forall i\ \mathit{schedule}(X^i, A^i) = \text{none})$$

In our algorithm, X is the set of operations already scheduled in the state currently under consideration. The operation x_j^k returned by the scheduler is an additional instruction to be scheduled in the same state. The primary restriction imposed by Constraint 4.2 is that the scheduler is a function of operations available in the state being scheduled. This constraint is weak because the set of available operations provides global information about the program—the scheduler can choose any statement that could be legally scheduled in the current state.[2] This particular constraint also has a significant design benefit: it cleanly separates the scheduler from the rest of the algorithm, thus isolating the most machine-dependent portion of the code. Any scheduler satisfying Constraint 4.2 will work with the software pipelining algorithm. In Section 6 we show how to generalize Constraint 4.2 to include resource constraints.

The scheduler is used by the software pipelining algorithm to repeatedly select operations for scheduling in a state. When the scheduler returns "none", the state is finished and successors of the state are scheduled. In Section 3 we presented a simplified example in which the scheduler chooses a subset of the available operations for scheduling. However, the iterative method described here is necessary in general because the operations available for scheduling in a state n can depend on the set of operations already scheduled in n. For example, consider the simple program fragment in Figure 5. Assuming that the parallel machine performs reads before writes, it is clear that both a and b can be scheduled together in the first (and only) state n_0. However, b cannot be scheduled in n_0 unless a is also scheduled in n_0—that is, b is not available for scheduling in n_0 unless a is scheduled in n_0. Otherwise, if the set of available operations were simply $\{a, b\}$, then the scheduler could choose to schedule b in n_0 and a in n_0's successor, which is incorrect.

A second constraint is placed on the available operations. At any moment there is a set of operations A that are available for scheduling associated with a state n. There are two ways that A can be updated. First, the procedure call $\mathit{update\text{-}one}(n, A, x^i)$ returns a pair consisting of the updated state with operations $\mathit{ops}(n) \cup \{x^i\}$ and the new set of available operations given that x^i has been scheduled. Second, when n is complete we wish to compute the set of available operations in the successors of n. The procedure call $\mathit{next}(n, A)$ maps n and A to a set of pairs $\{\langle n_j, A_j \rangle\}$ where for every branch $c_j \in \mathit{branch}(n)$, n_j is a new (empty) node, $n_j = \mathit{succ\text{-}on\text{-}branch}(n, c_j)$, and A_j is the set of operations available in n_j. Implementations of procedures *update-one* and *next* are given in Section 5.

[2]One can imagine even more powerful schedulers; for example, a scheduler having global information about not just one state at a time, but all states at all times. Because scheduling is inherently a very hard problem, however, it is not clear that this extra theoretical power translates into any practical advantage over the scheme presented here; see Section 8.

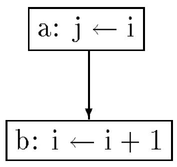


Figure 5: A simple program.

Constraint 4.3 Consider an arbitrary set of available operations A, state n, and operation x. Then there exists a set of operations B such that

$$\forall i \; \mathit{update\text{-}one}(m, A^i, x^{j+i}) = \langle m', B^i \rangle \quad \text{where} \quad \begin{array}{l} \mathit{ops}(m) = \mathit{ops}(n)^i \text{ and} \\ \mathit{ops}(m') = \mathit{ops}(n)^i \cup \{x^{j+i}\} \end{array}$$

Furthermore, there exist sets of operations A_j and states n_j for $1 \leq j \leq |\mathit{branch}(n)|$ such that

$$\forall i \; \mathit{next}(m, A^i) = \{\langle n_j, A^i_j \rangle\} \quad \text{where} \quad \mathit{ops}(m) = \mathit{ops}(n)^i$$

Constraint 4.3 says that the operations available may depend on which operations have already been scheduled and the relative distance (in iterations) between operations already scheduled, but it cannot depend on the actual values of the iterations of operations already scheduled. In the implementation of *update-one*, the result node is simply n updated to include the operation x^{j+i} (see Section 5.2). Whether Constraint 4.3 is satisfied or not depends on the form of the data dependence analysis used to maintain operation availability information. Standard data dependence graphs satisfy Constraint 4.3, as do extensions to dependence graphs, such as labeling edges with constant distance vectors [PBJ$^+$91]. In fact, as far as we know, every proposed representation of dependence information satisfies this constraint. Constraint 4.3 is needed to rule out pathological cases like Figure 4, where irregular dependence analysis leads to incorrect schedules.

4.2 The Algorithm

The software pipelining algorithm is given in Figure 6. Given an initial set of available operations, the procedure *pipeline* invokes the procedure *schedule-state* to build a single state, and then to build states for all the branches of that state, and so on. If at any point the algorithm encounters the same set of available operations (modulo iteration numbers) a second time, it uses the previously scheduled state. The algorithm never backtracks to explore alternative schedules. While a backtracking version could be designed easily, we feel a backtracking algorithm would be too slow to be practical.

The order in which *pipeline* processes the successors of a scheduled state is unspecified and makes no difference in the final parallel program. The order in which states are scheduled can make a difference in

```
procedure $schedule\text{-}state\ (n, A)$
    while $schedule(ops(n), A) \neq$ none do
        let $x = schedule(ops(n), A)$ in
            $\langle n, A\rangle \leftarrow update\text{-}one(n, A, x)$
    return $\langle n, A\rangle$

procedure $pipeline(A)$
    $\forall X\ scheduled\text{-}before[X] \leftarrow no$
    let $r$ be an empty node in $todo \leftarrow \{\langle r, A\rangle\}$
    while $\exists\langle n, A\rangle \in todo$ do
        if $\exists j$ s.t. $scheduled\text{-}before[A^j] \neq no$ then
            $n \leftarrow scheduled\text{-}before[A^j]$
            $todo \leftarrow todo - \{\langle n, A\rangle\}$
        else
            let $\langle n, A'\rangle = schedule\text{-}state\ (n, A)$ and
                $\{\ldots\langle n_i, A_i\rangle\ldots\} = next(n, A')$ in
                $scheduled\text{-}before[A] \leftarrow n$
                $todo \leftarrow (todo \cup \{\ldots\langle n_i, A_i\rangle\ldots\}) - \{\langle n, A\rangle\}$
```

Figure 6: The software pipelining algorithm.

the efficiency of the available operations computation; in Section 5 we present a slightly modified version of the algorithm in Figure 6 that processes states in an efficient order.

We use Constraints 4.2 and 4.3 to prove the correctness of the software pipelining algorithm in Figure 6. Let L be a sequential loop and let L' be the result of software pipelining. We show that $L \equiv L'$. As a first step in the proof, we must assume that the available operation analysis is correct. Intuitively, the available operation analysis is correct if any schedule that is consistent with the analysis preserves program semantics. We use the program in Figure 7 to formalize this intuition. This program is identical to the one in Figure 6 except that it does not reuse previously scheduled states. Let L_∞ be the (infinite) parallel program defined by this algorithm for a loop L. The available operation analysis is correct if for any choice of scheduler $L_\infty \equiv L$.

The essential step in proving the correctness of procedure *pipeline* is to show that every execution of L_∞ is also an execution of L'.

Lemma 4.4 Let $L' = pipeline(A)$ and let $L_\infty = pipeline2(A)$. For all k, if there is an execution $\langle\langle n_0, s_0\rangle, \ldots, \langle n_k, s_k\rangle\rangle$ of L_∞, then there is an execution $\langle\langle n'_0, s_0\rangle, \ldots, \langle n'_k, s_k\rangle\rangle$ of L'.

Proof: The proof is by induction on on the length of an execution. For the base case, let $e = \langle\langle n_0, s_0\rangle\rangle$ be an execution of L_∞. Consider how the initial states of L_∞ and L' are built. The initial set of available operations A is the same for both. Now, in procedure *pipeline* we have $scheduled\text{-}before[A] = no$, because initially no states are scheduled. Then $schedule\text{-}state\ (n, A) = \langle n_0, B\rangle$. Clearly $e = \langle\langle n_0, s_0\rangle\rangle$ is an execution of L'.

```
procedure pipeline2(A)
        let r be an empty node  in
            todo ← {⟨r, A⟩}
      (* Condition of the while is always true *)
      while ∃⟨n, A⟩ ∈ todo  do
            let ⟨n, A'⟩ = schedule-state (n, A) and
                {...⟨n_i, A_i⟩...} = next(n, A')  in
                todo ← (todo ∪ {...⟨n_i, A_i⟩...}) − {⟨n, A⟩}
```

Figure 7: An algorithm that defines an infinite parallel program.

For the induction step, assume that $e = \langle\langle n_0, s_0\rangle, \ldots, \langle n_i, s_i\rangle\rangle$ is an execution of L_∞ and that $e' = \langle\langle n'_0, s_0\rangle, \ldots, \langle n'_i, s_i\rangle\rangle$ is an execution of L'. Furthermore, assume that there exists a k such that when the states n_i and n'_i were scheduled, the sets of available operations were A in procedure *pipeline*2 and A^k in procedure *pipeline* respectively. Finally, assume that $ops(n_i) = ops(n'_i)^k$. It is easy to check that all of these assumptions hold after the base case.

If $n_i = \{stop\}$ then n_i and n'_i are final states and we are done. Otherwise, in the next transition we have

$$\begin{aligned}\delta(\langle n_i, s_i\rangle) &= \langle n_{i+1}, s_{i+1}\rangle\\ \delta(\langle n'_i, s_i\rangle) &= \langle n'_{i+1}, s_{i+1}\rangle\end{aligned}$$

The stores must be the same in the two transitions since, by hypothesis, n_i and n'_i have the same operations. Let c be the branch taken in state $\langle n_i, s_i\rangle$. Note c is also taken in state $\langle n'_i, s_i\rangle$, because n_i and n'_i have the same operations evaluated in the same store. To finish the proof, we need to show that n_{i+1} and n'_{i+1} have the same operations, possibly differing in iteration numbers used by the pipelining algorithm. That is, we must show that $ops(n_{i+1}) = ops(n'_{i+1})^j$ for some j.

Consider once more the state of the two software pipelining algorithms when n_{i+1} and n'_{i+1} are scheduled. By Constraint 4.3 and the induction hypothesis, $\langle m, B\rangle \in next(n_i, A)$ and $\langle m', B^k\rangle \in next(n'_i, A^k)$ where m and m' are fresh, empty states on branch c from n_i and n'_i respectively. Now there are two cases. For the first case, assume $\forall j$ *scheduled-before*$[B^j]$ = no when $\langle m', B^k\rangle$ is removed from the *todo* list by *pipeline*. In L_∞, let *schedule-state* $(m, B) = \langle n_{i+1}, C\rangle$. Then, by Constraints 4.2 and 4.3, in L' we have *schedule-state* $(m', B^k) = \langle n'_{i+1}, C^k\rangle$ and $ops(n_{i+1}) = ops(n'_{i+1})^k$.

For the second case, assume that, when $\langle m', B^k\rangle$ is removed from the *todo* list by *pipeline*, there is a j such that *scheduled-before*$[B^j] = n'$. Then n' was scheduled in L' using available operations B^j. The rest of the argument is symmetric to the case above, using B^j in place of B^k and the fact that $n'_{i+1} = n'$. □

Constraints 4.2 and 4.3 are needed to prove Lemma 4.4. These constraints ensure that having the same operations available for two states implies that all possible branches from those states are also be the same. Combining the correctness condition for the available operation analysis and Lemma 4.4 gives a proof of correctness.

Theorem 4.5 If procedure *pipeline* produces a loop L' from an initial loop L, then $L \equiv L'$.

Proof: To prove $L \equiv L'$ we must show $\forall s\ \mu(L,s) = \mu(L',s)$. First $\mu(L,s) = \mu(L_\infty, s)$, since by assumption the available operations analysis is correct. By Lemma 4.4 every execution of L_∞ is also an execution of L', so $\mu(L,s) = \mu(L',s)$. $\Box$

4.3 Termination

Theorem 4.5 proves that the software pipelining algorithm produces only correct results, but it does not show that it always terminates. To show termination, we must prove that the *todo* set in procedure *pipeline* is eventually empty. The *todo* set decreases in size when there is a pair $\langle n, A\rangle$ such that for some j, A^j has been scheduled previously. Let $\equiv$ be the equivalence relation on sets of operations $A \equiv B \Leftrightarrow \exists j\ s.t.\ A = B^j$. If we assume that the procedure *schedule-state* always terminates, then to prove termination it is sufficient to show that there are only finitely many equivalence classes under $\equiv$.

Unfortunately, there may be infinitely many equivalence classes and in fact the procedure *pipeline* is not necessarily terminating under the constraints given so far. Consider, for example, what happens if the A sets simply increase in size on each recursive call. A necessary condition for $A \equiv B$ is that $|A| = |B|$; if there are sets of unbounded cardinality, then there are infinitely many equivalence classes. An additional constraint is placed on the availability information to limit the size of the set of operations available for scheduling.

Constraint 4.6 There is a constant k such that for all possible availability sets A, if $x^j \in A$ then no $y^h \in A$ for any $h \geq j + k$.

This constraint states that operations can be available from at most k consecutive iterations at one time. Thus, the scheduler has a "sliding window" of operations and until operations in the first iteration are scheduled, the window cannot be shifted to include a new iteration at the end.

Lemma 4.7 Constraint 4.6 ensures that there are only finitely many equivalence classes of sets of operations under $\equiv$.

Proof: If there are n operations in a loop body and k consecutive iterations can appear in A, then every available operation set is a subset of $\{\ldots, x_i^{c+j_i}, \ldots\}$ for some c, $0 \leq i < n$, and $0 \leq j_i < k$. $\Box$

The value k of Constraint 4.6 is a parameter of the software pipelining algorithm. It need not be the same for every loop scheduled (i.e., it can be computed dynamically), but it must have a maximum value for any particular loop. Also, it is not necessary to make the window an integral number of iterations. Partial iterations work just as well, although the details of the implementation are a bit more complex.

While Constraint 4.6 is motivated by the need to guarantee termination, it also leads to a good implementation of the procedure *pipeline*. The most expensive part of *pipeline* is checking whether the current set of available operations A has ever been scheduled before for some A^j. For a window size of k iterations, operation availability information for iterations j through $j+k-1$ can be represented as a bit vector of length kn, where n is the number of operations in the sequential loop. The bit $hn+i$ is 1 if operation x_i^{j+h} is available for scheduling; otherwise it is 0. When iteration j has been completely scheduled (this occurs when the first n bits are all 0) the bit vector is shifted left n bits, discarding information for iteration j, and the last n bits are set to reflect the availability of operations in iteration $j+k$. With this representation, checking whether the same availability information has been seen before only requires checking whether the same bit vector has been seen before, which can be implemented very efficiently through hashing.

5 Available Operations

Available operations analysis plays a role in our algorithm similar to the role global data-flow analysis plays in traditional optimizing compilers. An algorithm for computing available operations was first given in [EN89] (for historical reasons, available operations were termed "unifiable-ops" in [EN89]). In this section we give a new presentation of available operations. While functionally equivalent to the algorithms of [EN89], our presentation is both simpler and more direct, and the final algorithms are easier to implement. The development is divided into two parts. First, we show how to compute the initial set of available operations. Second, we show how to incrementally update the information in response to decisions made by the scheduler. At the end of the section we prove the correctness of the analysis and discuss some efficiency considerations.

5.1 Computing Available Operations

Recall that Constraint 4.6 forces the available operations to span no more than k iterations of a loop. Therefore, to compute the operations available for scheduling it is sufficient to examine at most k iterations of a loop. Since any number of unrolled iterations form a loopless (acyclic) program, we restrict the problem of computing available operations to an analysis of loopless programs.

Computing available operations requires the use of dependence analysis between operations. There are many variations on dependence analysis in the literature that satisfy our requirements (Constraint 4.3) and it is beyond the scope of this paper to include them here [KKP+81, FOW87, PBJ+91]. The algorithms in this section are presented using an abstract mechanism for dependence. By using a particular dependence analysis representation the algorithms can be made more efficient. We use the following definitions to model dependence analysis.

Definition 5.1 A location is either a memory address or a register. For operations x and y and sets of operations X we define:

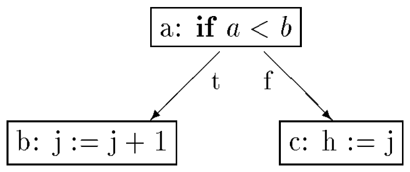


Figure 8: Operation b can kill a reference live at the root.

$$\begin{array}{rcl@{\qquad}rcl}
\mathit{write}(x) & = & \text{the set of locations } x \text{ may write} & \mathit{write}(X) & = & \bigcup_{x \in X} \mathit{write}(x) \\
\mathit{kill}(x) & = & \text{the set of locations } x \text{ must write} & \mathit{kill}(X) & = & \bigcup_{x \in X} \mathit{kill}(x) \\
\mathit{read}(x) & = & \text{the set of locations } x \text{ may read} & \mathit{read}(X) & = & \bigcup_{x \in X} \mathit{read}(x) \\
\mathit{depends}(x,y) & = & \mathit{write}(x) \cap (\mathit{read}(y) \cup \mathit{write}(y)) \neq \emptyset & \mathit{depends}(X,y) & = & \exists x \in X \text{ s.t. } \mathit{depends}(x,y)
\end{array}$$

The set $\mathit{write}(x)$ (resp. $\mathit{read}(x)$) must include every location x could ever write (resp. read). The set $\mathit{kill}(x)$ must include only locations x always writes. Two different sets $\mathit{write}(x)$ and $\mathit{kill}(x)$ are defined because dependence analysis must be conservative in general; it is not always possible to know at compile-time exactly which locations an operation may read or write. The predicate $\mathit{depends}(x, y)$ is true if there may be a dependence from x to y.

Defining correct available operation analysis requires identifying the operations that cannot be available because of potential data dependence violations. Assume that x precedes y on a path and $\mathit{depends}(x, y)$ is true. Then clearly y cannot be available on that path until x is scheduled, or else y could be scheduled before x, resulting in a dependence violation. The following dataflow equation specifies the operations reachable from state n that are not data dependent on an intervening operation:

$$\mathit{nodeps}(n) = \mathit{ops}(n) \cup \Big(\Big(\bigcup_{n' \in \mathit{succ}(n)} \mathit{nodeps}(n')\Big) - \{x \mid \mathit{depends}(\mathit{ops}(n), x)\}\Big)$$

Since the program fragment P being analyzed is loopless, $\mathit{nodeps}(n)$ can be computed for all n by a single bottom-up traversal of the control-flow graph for P.

The program in Figure 8 illustrates another situation in which an operation cannot be available. In this case, operation b cannot be available for scheduling in the first state, because its definition of j could change the value read by the reference to j in operation c. In standard compiler terminology, location j is *live* at the first state, and b can *kill* c's reference to j. Clearly, any operation that can kill a live reference cannot be available.

The second component of the available operations analysis is a computation of live references. A reference to location l is live at a state n if there is a state reachable from n where l is (potentially) read and there is no intervening write of l. A conventional live reference analysis is not sufficient for our purposes; instead, we wish to compute live references discounting the effect of a particular operation x. More precisely, we wish to know the set of live references assuming that x has been moved to the root state of the program. In this case, to say "x has been moved to the root" means that all occurrences of

x that can potentially move to the root are not counted in the live variable computation. The intuitive justification behind this computation is that when moving an operation x in the schedule, it is necessary to check if x will kill live references in its new position. However, in deciding whether or not x will kill live references in its new position one should not count references of x itself in its current position.

The following dataflow equation defines the set of locations live at state n modulo operation x:

$$
\begin{aligned}
\mathit{live}(n, x) \;=\;& \mathit{read}(Y) \cup \Big(\Big(\bigcup_{n' \in \mathit{succ}(n)} Z_{n'}\Big) - \mathit{kill}(Y)\Big) \\
& \text{where } Z_{n'} = \begin{cases} \mathit{live}(n', x) & \text{if } \; x \in \mathit{nodeps}(n') \\ \mathit{live}(n', \mathit{stop}) & \text{otherwise} \end{cases} \\
& \text{where } Y = \mathit{ops}(n) - \{x\}
\end{aligned}
$$

The two cases in the definition of $Z_{n'}$ distinguish between the cases where occurrences of x can or cannot be blocked by data dependencies. If there is an occurrence of x on a path that is not blocked by data dependencies (i.e., $x \in \mathit{nodeps}(n')$) then that occurrence of x is discounted in the live reference computation (i.e., $\mathit{live}(n', x)$). If there is no occurrence of x that can potentially move, then all live references are counted (i.e., $\mathit{live}(n', \mathit{stop})$ counts all references, since *stop* has no effect on the store). As with the computation of *nodeps*, $\mathit{live}(n, x)$ can be computed for all states n and operations x by a single bottom-up traversal of the control-flow graph for P. Some further improvements to the efficiency of this procedure are discussed at the end of the section.

Let r be the initial state (or *root*) of P. An operation x is available for scheduling in r if it satisfies three conditions: x is in $\mathit{nodeps}(r)$, x does not kill any live reference in operations other than x, and x is in the "sliding window" of operations. Recall that Constraint 4.6 requires that operations from no more than k consecutive iterations be available for any state. For a set of available operations A, let $\mathit{min\text{-}it}(A)$ be the minimum iteration number of any operation in A.

$$
\begin{aligned}
\mathit{available}(r) \;=\;& \{x^i \mid x^i \in R \text{ and } i - \mathit{min\text{-}it}(R) < k\} \\
& \text{where } R = \mathit{nodeps}(r) - \{x \mid \mathit{write}(x) \cap \mathit{live}(r, x) \neq \emptyset\}
\end{aligned}
$$

Note that we are concerned with live references only at the root r; operations that potentially kill live references at an internal state n are included in $\mathit{nodeps}(n)$. The program in Figure 9 illustrates this situation. Unlike Figure 8, the reference to j in operation c is not live at the root because it reads the value written by d. The important observation is that any operation that can kill a reference that is not live at the root (e.g. b can kill c's reference to j) must be dependent on some preceding operation (e.g. $\mathit{depends}(d, b)$). That is, a reference to j that is not live at the root must be preceded by an operation that writes j; this operation prevents other operations that could write j from being available at the root.

The most expensive part of computing $\mathit{available}(r)$ is computing $\mathit{live}(r, x)$ for every operation x. The efficiency of the naive procedure described above can be improved in two ways. First, it is not necessary to compute $\mathit{live}(r, x)$ for every x; it is sufficient to compute it only for those operations in $\mathit{nodeps}(r)$, because $\mathit{nodeps}(r)$ is a superset of the operations available for scheduling. Second, operations that kill

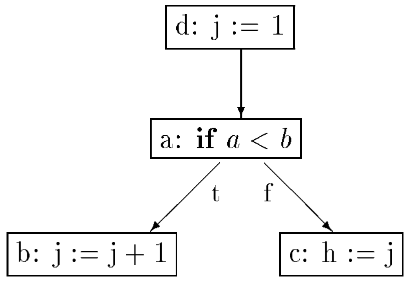


Figure 9: Operation b can kill j, but j is not live at the root.

live references could be detected earlier in the computation instead of checking only at the root; we have not given this alternative to simplify the presentation.

5.2 Maintaining Available Operations

We first describe at a high level how available operations are maintained, after which we give implementations of the procedures *next* and *update-one*. Let P be a sequential loopless program. We add an empty state r (a state with no operations) to P and make it the root; r is the initial state in procedure *pipeline* (see Figure 6). This empty state r will be filled with operations chosen by the scheduler.

The next step is to compute the dataflow analysis of Section 5.1. At this point the set of operations available for scheduling in state r is $\mathit{available}(r)$.

Once the initial global analysis of P is completed, we are ready to begin scheduling states. When state n is scheduled, it is first filled with operations (by *schedule-state*), and then $\langle n, A\rangle$ is removed from the *todo* set and n's successors are added to the *todo* set. A state in the *todo* set is a *frontier* state. At any point in the incremental development of the parallelized loop, every frontier state of the parallel loop has the property that its known predecessors have been scheduled and its successors have yet to be scheduled. (The "unknown" predecessors are those that are added through backedges inserted that complete the software pipelined loop.) Available operations are needed only for the frontier states; predecessors of frontier states are never modified. When procedure *pipeline* terminates, there are no frontier states and the (modified) program P is the parallel loop.

Figure 10 gives a generic snapshot of the algorithm's data structures during scheduling. The states above the line labeled A are parallel states, already scheduled by the algorithm. These states are arranged as a tree, except where backedges have been added by pipelining. The states between the lines A and B are the frontier states. These are empty states that have yet to be filled with operations. The states below line B are states of the original sequential loop. Conceptually, these states are not part of the parallel loop; they are used instead to compute the available operations information for the frontier states.

The first *todo* set is $\{\langle r, \mathit{available}(r)\rangle\}$. Thus, initially r is the only frontier state. The procedure *pipeline* selects a pair $\langle r, \mathit{available}(r)\rangle$ from *todo* and fills it with operations by calling $\mathit{schedule\text{-}state}(r, \mathit{available}(r))$. The procedure *schedule-state* in turn calls *update-one* one or more times to choose the scheduled oper-

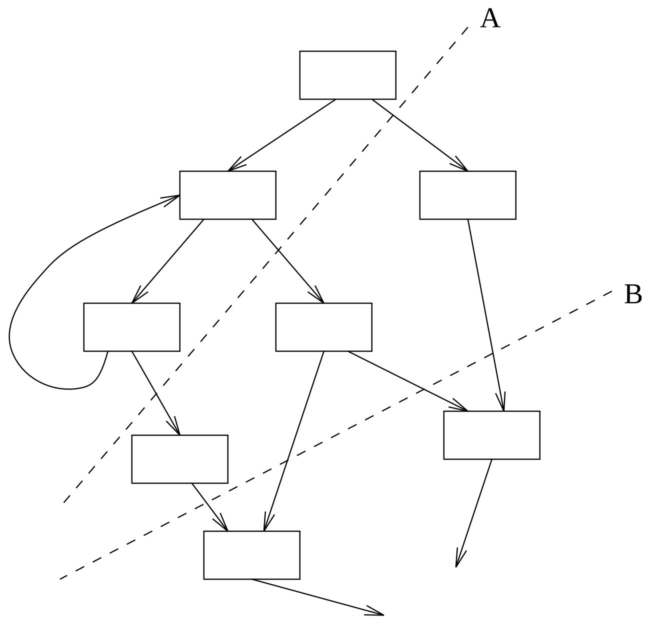


Figure 10: A snapshot of software pipelining during scheduling.

ations (see Figure 6). The procedure call *update-one*(*available*(r), r, x) performs two tasks. First, x is deleted from interior states of P and x is added to the frontier state r. Thus, this transformation moves x to r from its original place in the sequential schedule. (Some copies of x may have to remain in interior states of P if x cannot move on all paths to the frontier state; see the discussion below.) When a test is moved to r the control flow of P must be modified to preserve P's semantics. Second, the sets *nodeps* and *live* are updated where necessary.

An important fact is that both the deletion of x and updating of the *nodeps* and *live* sets can be restricted to a relatively small subset of the states of P; this property makes the incremental cost of maintaining available operations reasonable. The new set of available operations is (the updated) *available*(r).

When the scheduling of r is complete, *next*(r, A) $= \{\langle r_i, A_i\rangle\}$ is the set of (empty) successors r_i of r and the corresponding sets of available operations A_i. We implement *next*(r, A) by inserting a new, empty state r_i before each *succ*(r, c_i) on branch $c_i \in$ *branch*(r). The set *available*(r_i) is exactly the set of operations available for scheduling on branch c_i from r. Note that the r_i are new frontier states of P. This implementation of *next* allows P (and therefore the available operation analysis) to be shared among all elements of the *todo* set. As scheduling proceeds there will be multiple frontier states in P, one for each element in *todo*. An implementation of *next* is given in Figure 11.

Lemma 5.2 Let P' be P with the modifications performed by *next*. Then $P' \equiv P$.

Proof: Procedure *next* only inserts empty nodes in P. $\Box$

To complete the description of available operations we must give an implementation of procedure *update-one*. We could do this trivially in terms of the local transformations of Percolation Scheduling

```
procedure $next(r, A)$
    for each $c_i \in branch(r)$ do
        let $r_i$ be a fresh empty state, $n = succ\text{-}on\text{-}branch(r, c_i)$ in
            $succ\text{-}on\text{-}branch(r, c_i) \leftarrow r_i$
            $succ\text{-}on\text{-}branch(r_i, \langle\rangle) \leftarrow n$
            $nodeps(r_i) \leftarrow nodeps(n)$
            $live(r_i, x) \leftarrow live(n, x)$ for all operations $x$
    return $\{\langle r_i, available(r_i)\rangle | r_i$ as defined above$\}$
```

Figure 11: Implementation of *next*.

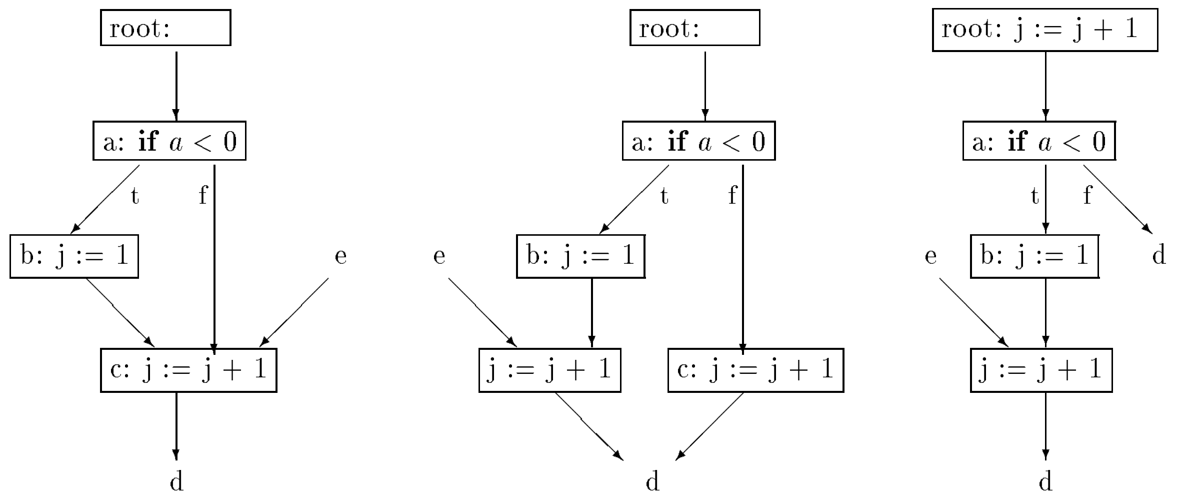
[Nic85], but for completeness we describe a direct implementation that is closer to the way it should be done in practice. Let r be a frontier state of P, let $x = schedule(ops(r), available(r))$, and assume that $x \neq$ none. We first describe how x is deleted from P and added to r when x is an assignment; this is the easier case. Moving an operation x while preserving P's semantics is a little subtle because x may be available at the frontier state but still blocked by data dependencies on some paths. The program in Figure 12a illustrates this situation. In this example, c is available at the root because c does not kill any references live at the root and because there is a path from the root to c (in this case passing through the false branch of a) such that c is not dependent on any operation on the path. However, there may be other paths from the root to c (in this case passing through the true branch of a) such that c is dependent on some operation on the path; clearly, c cannot be deleted from such a path. In addition, there may even be paths from other frontier states to c (represented by the incoming edge from e). If c is moved to *root* in Figure 12 it still must be preserved on paths from other frontier states.

As illustrated in Figure 12b, this problem can be resolved by duplicating states so that no instance of the operation being moved is shared between paths where it can move and paths where it cannot move. In this example, only the single state containing c needs to be duplicated, but in general multiple states may have to be duplicated. In Figure 12c, state c has been deleted and the operation moved to the root. It is easy to verify that the program in Figure 12c is equivalent to the program in Figure 12a.

To formalize which states are duplicated and which states are deleted we need some additional definitions and notation. A *path* is a sequence of states $\langle n_1, \ldots, n_k\rangle$ such that $n_{i+1} \in succ(n_i)$ for all $1 \leq i < k$. A state n_k is *covered* by a state n_1 for operation x if there is a path $\langle n_1, \ldots, n_k\rangle$ such that operation x is in $nodeps(n_i)$ for every n_i on the path:

$$covered(n_1, x) = \{n_k | \text{ there exists is a path } \langle n_1, \ldots, n_k\rangle \text{ s.t. } \forall i, 1 \leq i \leq k \; x \in nodeps(n_i)\}$$

We say a path is *covered by* (n, x) if every state of the path is in $covered(n, x)$. When an operation x is moved to a frontier state r it should be deleted only from paths that are covered by (r, x); other paths should be left unchanged. The simplest case is if every path to x is covered by (r, x). We say a loopless program P is *delete consistent for* (r, x) if for every $n \in covered(r, x)$ such that $n = \{x\}$, every path from a frontier state to n is covered by (r, x). If P is delete consistent for (r, x), then x is not blocked by data



(a) Before c is moved. (b) Delete consistent. (c) After c is moved.

Figure 12: Moving an assignment.

dependencies on any path to the frontier state r. Hence, we can delete states $n = \{x\}$ in *covered*(r, x), update the predecessors of n to point to n's successor, and add x to r.

Lemma 5.3 Let x be an assignment such that $x \in$ *available*(r) and let $N = \{n | n = \{x\}$ and $n \in$ *covered*$(r, x)\}$. Assume that P is delete consistent for (r, x). Let P' be P with the following changes. (Recall that $\langle\rangle$ is the empty branch—see Section 2.)

- modify each n' where *succ-on-branch*$(n', c) = n$ for some $n \in N$ so that *succ-on-branch*$(n', c) =$ *succ*$(n, \langle\rangle)$
- delete every $n \in N$
- $r \leftarrow r \cup \{x\}$

Then $P \equiv P'$.

Proof: For brevity we only sketch the proof. The transformation can be implemented by a sequence of semantics-preserving Percolation Scheduling transformations between adjacent nodes [Nic85]. Since each individual Percolation Scheduling transformation preserves program semantics, the entire sequence preserves program semantics. □

Of course, Lemma 5.3 only applies if P is delete consistent. We next show how to make an arbitrary loopless program delete consistent for (r, x). The set of *predecessors* of a state n is *pred*$(n) = \{n' | n \in$ *succ*$(n')\}$. The following lemma gives an easy test for determining whether P is delete consistent.

Lemma 5.4 P is delete consistent for (r, x) iff $n \in$ *covered*$(r, x) \Rightarrow$ *pred*$(n) \subseteq$ *covered*(r, x).

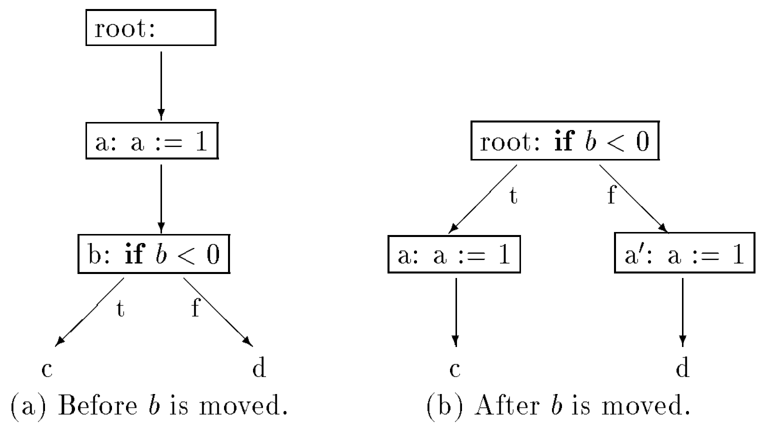


Figure 13: Moving a test.

Proof: If every predecessor of a member of $\mathit{covered}(r, x)$ is in $\mathit{covered}(r, x)$, then clearly every path from r to $n \in \mathit{covered}(r, x)$ is covered by (r, x). For the other direction, assume that there is an $n \in \mathit{covered}(r, x)$ and for some $n' \in \mathit{pred}(n)$, $n' \notin \mathit{covered}(r, x)$. Then there must be a path from some frontier state r' of the form $\langle r', \ldots, n', n\rangle$. This path is not covered by (r, x). $\Box$

The following algorithm makes P delete consistent for (r, x). Let $C = \mathit{covered}(r, x)$. Iterate the following two steps until no n is chosen in (1):

(1) Choose $n \in C$ such that some p in $\mathit{pred}(n)$ is not also in C.

(2) Let n' be a duplicate of n and for every $p \in \mathit{pred}(n)$ such that $p \notin C$, if $\mathit{succ\text{-}on\text{-}branch}(p, c) = n$ then modify p so that $\mathit{succ\text{-}on\text{-}branch}(p, c) = n'$.

Note that this algorithm copies the minimum number of states needed to make P delete consistent.

Once P is delete consistent the steps of Lemma 5.3 can be applied to move x to the frontier state. All that remains is to update the *nodeps* and *live* sets. States that are duplicated in making P delete consistent retain the *nodeps* and *live* information of the original state. The set of states for which the analysis can change is $\mathit{covered}(r, x)$. Since both *nodeps* and *live* are computed bottom-up, the analysis can be updated in a single bottom-up pass over the paths covered by (r, x).

Finally, we show how to update the available operations in the case where the instruction chosen by *update-one* is a test. Let r be a frontier state of P, and let $x = \mathit{schedule}(\mathit{ops}(r), \mathit{available}(r))$. For *update-one* to move a test x while preserving P's semantics, it is necessary to modify the control flow of P. Intuitively, we duplicate all the covered paths from r to x; the original set of paths leads to the successor on x's true branch, the duplicate set of paths leads to the successor on x's false branch. This transformation is illustrated in Figures 13. The program in Figure 13a is already delete consistent for (root, b). When b is moved to *root* in Figure 13b, state a is duplicated on b's *true* and *false* branches to preserve control flow. Recall that a branch c is a truth assignment to tests $\langle x_1 = b_1, \ldots, x_n = b_n\rangle$ where b_i is one of *true* or *false*. The following lemma shows how to move a test to a frontier state.

Lemma 5.5 Let P be a program with frontier state r and let $x \in \mathit{available}(r)$ where x is a test. Assume P is delete consistent for (r, x) and let $X = \mathit{covered}(r, x)$ in P. Program P' is P with the following modifications performed in order:

(1) For each $n \in X$, let n' be a state such that $\mathit{ops}(n) = \mathit{ops}(n')$ and for each $c \in \mathit{branch}(n)$

$$\mathit{succ\text{-}on\text{-}branch}(n', c) = \begin{cases} n_1 & \text{if } n_1 \notin \mathit{covered}(r, x) \\ n_1' & \text{if } n_1 \in \mathit{covered}(r, x) \text{ and } \mathit{ops}(n_1) \neq \{x\} \\ \mathit{succ\text{-}on\text{-}branch}(n_1, \langle x = \mathit{false} \rangle) & \text{if } n_1 \in \mathit{covered}(r, x) \text{ and } \mathit{ops}(n_1) = \{x\} \end{cases}$$

where $n_1 = \mathit{succ\text{-}on\text{-}branch}(n, c)$.

(2) For each $n \in X$, if $\mathit{succ\text{-}on\text{-}branch}(n, c) = n_1$ and $\mathit{ops}(n_1) = \{x\}$, then modify n so that $\mathit{succ\text{-}on\text{-}branch}(n, c) = \mathit{succ\text{-}on\text{-}branch}(n_1, \langle x = \mathit{true} \rangle)$

(3) For each $c \in \mathit{branch}(r)$ where $c = \langle x_1 = b_1, \ldots, x_n = b_n \rangle$ and $n = \mathit{succ\text{-}on\text{-}branch}(r, c)$ do

$$\begin{aligned} \mathit{succ\text{-}on\text{-}branch}(r, \langle x = \mathit{true}, x_1 = b_1, \ldots, x_n = b_n \rangle) &= n \\ \mathit{succ\text{-}on\text{-}branch}(r, \langle x = \mathit{false}, x_1 = b_1, \ldots, x_n = b_n \rangle) &= \begin{cases} n & \text{if } n \notin X \\ n' & \text{if } n \in X \end{cases} \end{aligned}$$

(4) Let $\mathit{ops}(r) \leftarrow \mathit{ops}(r) \cup \{x\}$

Then $P \equiv P'$. [3]

Proof: Again, for brevity we only sketch the proof. It is easy to verify that P' preserves the control flow of P. As in the proof of Lemma 5.3, the transformation can be expressed as a sequence of local transformations between adjacent nodes. □

Part (1) of Lemma 5.5 duplicates covered paths by creating a copy n' of every state in $\mathit{covered}(r, x)$ and by assigning successors so that the paths formed by the n' lead to the false branch of x. Part (2) modifies the original states in $\mathit{covered}(r, x)$ so that they lead to the true branch of x. Part (3) modifies the branches of r to point to the original nodes if x is true and to the copied nodes if x is false. A description of procedure *update-one* is given in Figure 14.

The implementation we have described is somewhat naive and there are inefficiencies that can be eliminated at the cost of greater complexity in the algorithm. Most of the potential problems are related to space explosion, either in the size of the final code or in the size of intermediate data structures used by the algorithm. Some states that are initially different may become identical as a result of scheduling operations. This observation applies to both states in the parallel schedule and states that have yet to be scheduled. A good implementation should merge states that are identical and are on identical paths. When performed on the states of the parallel schedule, this optimization reduces the size of the final code.

[3] Note that in this construction the original states containing x are not removed from P, but they become unreachable because control-flow is redirected around them.

```
procedure update-one($r, A, x$)
    make $P$ delete consistent for $(r, x)$ (Lemma 5.4)
    $X \leftarrow covered(r, x)$
    if $x$ is an assignment then
        perform steps in Lemma 5.3
    else
        perform steps in Lemma 5.5
    update nodeps and live for $n \in X$ and any states added by Lemma 5.5
    return $(\langle r, available(r)\rangle)$
```

Figure 14: Implementation of procedure *update-one*.

A separate potential problem lies in the definition of delete consistency. Making the sequential program delete consistent prior to moving an operation x may result in duplicating many states of the sequential program. These duplicates cannot subsequently be merged because x occurs on one set of paths in its original position (i.e., on those paths where x was blocked by a data dependence) and not on the set of paths where x was moved. A partial solution is to move x as far as possible on the paths where it is blocked by a data dependence, thus allowing some sharing of common paths. Some scheduling systems have this property [Nic85, ME92]. However, this optimization may be of marginal value in our algorithm, because of the duplicated states of the sequential program are soon eliminated by subsequent scheduling anyway.

Another approach to improving the efficiency of the techniques presented here is to use a representation other than the control-flow graph for computing available operations. The obvious alternative is to use some form of the program dependence graph, which admits more efficient algorithms for some purposes (see [LA92, AJLS92] for uses of program dependence graphs in the context of software pipelining). We have presented our techniques using a control-flow graph representation for simplicity only—there is no barrier to using other, potentially faster, representations in an implementation.

5.3 Correctness of the Analysis

In Section 4, we assumed the available operations analysis was correct to prove the correctness of the software pipelining algorithm. Recall that the available operations analysis is correct if $L_\infty \equiv L$, where L is a sequential loop and L_∞ is the infinite parallel program computed by *pipeline2*. In this section we prove that the implementation of available operations given in Sections 5.1 and 5.2 is correct.

Lemma 5.6 Let L_∞ be the program defined by *pipeline2* for some loop L. Then for any scheduler, $L_\infty \equiv L$.

Proof: Let P_∞ be the (infinite) acyclic program formed by full unrolling of L. Apply *pipeline2* using P_∞ for the available operations analysis and let L_∞ be the final program. Each transformation of P_∞ by *next* or *update-one* preserves the semantics of P_∞ by Lemmas 5.2, 5.3, and 5.5. Therefore, $L \equiv P_\infty \equiv L_\infty$. $\square$

5.4 Managing the Window

For performance reasons, it is obviously desirable to minimize the number of iterations of L that are actually used in the available operations analysis. It is possible to use only a few iterations of P because Constraint 4.6 forces the available operations analysis for any state to span no more than k iterations of loop L. In this section we show that the number of iterations needed for available operations analysis can be limited to k.

The only problem with limiting the number of iterations used in the analysis is that different frontier nodes may require available operations from different iteration windows. For example, for a frontier state r operations may be available from iterations i to $i+k-1$, but for another frontier state r' operations may be available from iterations $i+c$ to $i+k+c-1$. In a naive implementation, P must contain operations from iterations i through $i+k+c-1$ to cover both frontier states. Fortunately, this is not necessary. We can first schedule r using iterations i to $i+k-1$ and any other states that have operations available from iteration i. Once all states with operations available from iteration i are scheduled, a new iteration of L can be added to P and the window shifted to $i+1$ to $i+k$.

Figure 15 is a modified version of *pipeline*. In this implementation, all frontier states that have operations available from iteration i are scheduled before any frontier states that have operations available from iteration $i+1$. A new iteration is added to P only when every state that has operations available from iteration i is already scheduled. Thus, P always contains the minimal number of iterations and iterations are added to P as infrequently as possible.

There is one detail omitted from Figure 15. When the ith iteration of L is added to P, the *live* sets of the leaf states (i.e. the states at the end of iteration i) must be initialized to the set of locations live at the end of iteration i.

6 Resources

Resource allocation is a critical issue for software pipelining algorithms. In this section, we show how the allocation of functional units can be smoothly integrated into our software pipelining algorithm. Our approach to incorporating functional resources is similar to the reservation table methods used in dynamic scheduling algorithms [Bae80]. To describe the modifications to the algorithm that accommodate functional resources we require some additional definitions. Let $\{f_1, \ldots, f_n\}$ be the set of functional units for a machine. We drop the assumption that every operation executes in a single cycle and assume that c is the greatest number of cycles required by any operation. A *reservation table* is an $n \times c$ array of boolean values, where entry (i, j) is *true* iff resource f_i is busy at cycle j. For an operation x, the reservation table *resources*(x) describes the resources required by x in each cycle of x's execution.

There is one difficulty in extending the model to multi-cycle operations. If an operation x requires

```
procedure $pipeline(A)$
    $\forall X$ $scheduled\text{-}before[X] \leftarrow no$
    $P \leftarrow k$ iterations of L with empty root $r$
    $todo \leftarrow \{\langle r, A\rangle\}$
    $itnum \leftarrow 0$
    while $todo \neq \emptyset$ do
        begin
            while $\exists \langle n, A\rangle \in todo$ such that $min\text{-}it(A) = itnum$ do
                begin
                    if $\exists j$ s.t. $scheduled\text{-}before[A^j] \neq no$ then
                        $n \leftarrow scheduled\text{-}before[A^j]$
                        $todo \leftarrow todo - \{\langle n, A\rangle\}$
                    else
                         let $\langle n, A'\rangle = schedule\text{-}state\ (n, A)$ and
                            $\{\ldots\langle n_i, A_i\rangle\ldots\} = next(n, A')$ in
                            $todo \leftarrow (todo \cup \{\ldots\langle n_i, A_i\rangle\ldots\}) - \{\langle n, A\rangle\}$
                            $scheduled\text{-}before[A] \leftarrow n$
                end
            $itnum \leftarrow itnum + 1$
            add one iteration of $L$ to end of $P$ and update analysis
        end
```

Figure 15: The modified software pipelining algorithm.

multiple cycles to complete, then its result is not available for multiple cycles. However, data dependencies and resource constraints alone do not prevent operations that depend on x's result from being scheduled in the cycle after x is initiated. We resolve this problem by treating an i-cycle operation x as i one-cycle operations; operations that depend on the result of x are dependent on the last operation in the chain. To guarantee legal schedules, it is necessary to constrain the i unit-cycle operations to be scheduled in successive cycles without interruption. This constraint can be encapsulated entirely within the policy for selecting operations to schedule and thus does not affect the overall structure of the software pipelining algorithm.

To allocate functional units, the software pipelining algorithm is modified so that when a state n is scheduled there is a reservation table associated with n describing resource usage at that point in the schedule. The scheduler is modified so that it chooses an operation that is both available and for which resources can be allocated. Two reservation tables R_1 and R_2 are *compatible* if they do not require the same functional unit in the same cycle; i.e. there is no point (i, j) such that $R_1(i, j) = \mathit{true} = R_2(i, j)$. If the reservation table R is associated with state n, then the scheduler must choose an operation x to schedule in n such that $\mathit{compatible}(R, \mathit{resources}(x))$. The following constraint modifies Constraint 4.2 to include reservation tables.

Constraint 6.1 Let X and A be sets of operations and let R be a reservation table. The scheduler is a function that takes a set of already scheduled operations, available operations, and reservation table and returns an operation. In addition, if $X = \emptyset$ and there exists an $x \in A$ such that $\mathit{compatible}(\mathit{resources}(x), R)$, then $\mathit{schedule}(\mathit{ops}(n), A, R) \neq$ none (i.e. the scheduler must choose an operation in every state if possible). Finally, we require that

$$(\exists x_j \; \forall i \; \mathit{schedule}(X^i, A^i, R) = x_j^{k+i},\ x_j^{k+i} \in A^i, \text{ and } \mathit{compatible}(\mathit{resources}(x_j^{k+i}), R)) \ \bigvee$$
$$(\forall i \; \mathit{schedule}(X^i, A^i, R) = \text{none})$$

The procedures *next* and *update-one* must also be modified to update reservation tables to reflect the changes in available resources when operations are scheduled. The procedure call $\mathit{next}(n, A, R)$ should advance the reservation table R by one cycle to reflect the fact that in successors of n the resources used in the first cycle of R are no longer reserved. The procedure call $\mathit{update\text{-}one}(n, A, R, x)$ should not only update n and A, but also update R by adding the resources required by x.

The next constraint modifies Constraint 4.3 to include reservation tables. The logical or of two reservations tables $R_1 \vee R_2$ is a table R such that $R(i, j) = R_1(i, j) \vee R_2(i, j)$. The reservation table $\mathit{advance}(R)$ is a table R' such that $R'(i, j) = R(i, j + 1)$ for $j < c$ and $R(i, c) = \mathit{false}$.

Constraint 6.2 Consider an arbitrary set of available operations A, state n, operation x, and reservation table R. Then there exists a set of operations B such that

$$\forall i \; \mathit{update\text{-}one}(m, A^i, R, x^{j+i}) = \langle m', B^i, R \vee \mathit{resources}(x^j) \rangle \quad \text{where} \quad \begin{array}{l} \mathit{ops}(m) = \mathit{ops}(n)^i \text{ and} \\ \mathit{ops}(m') = \mathit{ops}(n)^i \cup \{x^{j+i}\} \end{array}$$

```
procedure $schedule\text{-}state\ (n, A, R)$
    while $schedule(ops(n), A, R) \neq$ none do
        let $x = schedule(ops(n), A, R)$ in
            $\langle n, A, R\rangle \leftarrow update\text{-}one(n, A, R, x)$
    return $\langle n, A, R\rangle$

procedure $pipeline(A)$
    $\forall X, R\ scheduled\text{-}before[X, R] \leftarrow no$
    let $r$ be an empty node, $R$ be an empty reservation table in
        $todo \leftarrow \{\langle r, A, R\rangle\}$
        while $\exists\langle n, A, R\rangle \in todo$ do
            if $\exists j$ s.t. $scheduled\text{-}before[A^j, R] \neq no$ then
                $n \leftarrow scheduled\text{-}before[A^j, R]$
                $todo \leftarrow todo - \{\langle n, A\rangle\}$
            else
                let $\langle n, A', R'\rangle = schedule\text{-}state\ (n, A, R)$ and
                    $\{\langle n_i, A_i, R''\rangle\} = next(n, A', R')$ in
                    $todo \leftarrow (todo \cup \{\langle n_i, A_i, R''\rangle\}) - \{\langle n, A, R\rangle\}$
                    $scheduled\text{-}before[A, R] \leftarrow n$
```

Figure 16: The software pipelining algorithm.

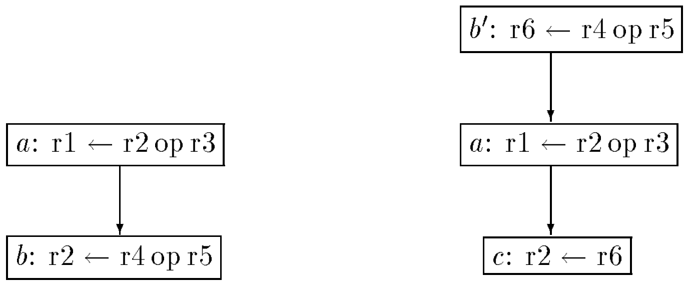
Furthermore, there exist sets of operations A_j and states n_j for $1 \leq j \leq |branch(n)|$ such that

$$\forall i\ next(m, A^i, R) = \{\langle n_j, A^i_j, advance(R)\rangle\} \quad \text{where} \quad ops(m) = ops(n)^i$$

Figure 16 gives a modified version of the software pipelining algorithm that includes reservation tables. For simplicity, the modifications are presented to the original algorithm in Figure 6 rather than the more efficient version in Figure 15. Note that the detection of repeating states now involves both the set of available operations and the reservation table. Using Constraints 6.1 and 6.2, it is straightforward to adapt the original proof of correctness of the software pipelining to prove the correctness of the algorithm in Figure 16. Termination is still guaranteed because there are only a finite number of reservation tables and therefore repeating states are guaranteed to occur.

6.1 Register Allocation

Registers are another critical resource that must be utilized effectively to achieve good results in practice. Traditional register allocation can interact very badly with software pipelining. If register assignment is performed before scheduling—the usual practice—then software pipelining may produce poor results, because the register allocator may unnecessarily reuse registers, thus adding data dependences to the program. Our approach is to modify an initial register allocation "on the fly" during software pipelining. The basic technique is easy to describe; it is based on a similar technique of Ebcioğlu[Ebc87]. Consider



(a) Instruction b is unavailable. (b) After renaming registers.

Figure 17: Dynamically improving register allocation.

the program fragment in Figure 17(a). In this example, operation b is not available for scheduling at the root because its target register is the one of the operand registers of operation a. However, if there is a spare register, then the dependence can be broken by renaming the destination register of b as in Figure 17(b). Now operation b' is available for scheduling. It is necessary to insert a register move c into the program to restore the machine state after operation a. This transformation is a heuristic—it assumes that the advantage gained in eliminating the dependence outweighs the cost of the extra copy. This is usually true, and almost always the copy operation can be removed by a later global pass of generalized copy propagation [PNW92].

There is an additional problem with the register allocation scheme described above. Including register allocation in the software pipelining algorithm requires that registers be taken into account when determining when two states are "the same". A sufficient condition is that two states s and s^i can be considered the same only if each register holds a value generated by operation x in state s and a value generated by operation x^i in state i. This condition guarantees correctness and termination and is analogous to the similar requirements for available operations and functional units.

In practice it appears that this condition alone permits an impractical number of states to be generated in some loops and a repeating pattern fails to emerge within a reasonable number of steps. The problem is that, even for small register files, the number of possible assignments of values to registers is astronomical. To accelerate convergence of the pipelining algorithm, it is necessary to limit the space of possible register assignments in some way. The solution we use is as follows. A register file r is a *renaming* of register file r' if r can be mapped to r' by some set of register-to-register transfers. In the software pipelining algorithm, two states are considered to be equivalent if the register files in the two states are renamings of each other and the other conditions (on operations and functional units) are satisfied.

This design makes identifies together many register files, thereby accelerating convergence of the algorithm. The cost is that register-to-register transfers must be issued on the backedges of pipelined loops to move values into the correct registers. These copy operations can be eliminated by a separate copy elimination optimization pass after software pipelining. Leaving the copy operations in the code is reasonable as well, as they incur only a minor performance penalty (see Section 7).

7 Implementation and Experiments

The software pipelining algorithm described here has been implemented as part of a compiler project at the University of California, Irvine. The compiler is a version of the GNU C compiler (GCC) modified to accommodate our methods. GCC is used as a front-end to translate the C level source into an intermediate representation. This translation includes an initial register allocation and a number of common optimizations: constant folding, jump optimization (e.g., removing jumps to jumps), common subexpression elimination, and strength reduction. GCC's instruction scheduling, loop unrolling, and inlining are disabled and replaced by our software pipelining and scheduling algorithm.

A number of incremental optimizations (e.g. incremental tree-height reduction) are beneficial in conjunction with software pipelining pipelining. For the results presented in this paper, only dynamic renaming (see Section 6.1) and *load-after-store elimination* are performed together with pipelining. Load-after-store elimination identifies loads that depend on a unique store; such loads can be eliminated in favor of uses of the value being stored. (In some cases, the store can be removed also if it is known that the eliminated load is the only read of the location written to by the store.) Load-after-store elimination is useful because it removes register spill code that becomes dead as a result of dynamic renaming. Both dynamic renaming and load-after-store elimination are an inherent part of our "on the fly" register allocation scheme.

The strength of our software pipelining framework is the flexibility to exploit whatever fine-grain parallelism is available in a loop. Restrictions placed on code motions are designed to be as weak as possible while still guaranteeing correctness and termination. As discussed below, this flexibility does in fact translate into very good speedups across a variety of architectural models.

The downside of the weak restrictions of our system is that there are a huge number of potential states, even for small loops and machines with modest resources. The huge state space can cause slow convergence of pipelining to a pattern and large final loops. There is a clean solution to this problem: the scheduler should be designed to minimize code explosion by restricting code motions that increase code size. For the purposes of this paper, we have focussed the experiments to reveal information about the software pipelining algorithm and not information about particular smart scheduling heuristics. Thus, we have used only very simple greedy list-scheduling heuristics that make no effort to take account of the impact of code motions on code size. As we shall see shortly, in the majority of cases the size of the state space is not a problem and software pipelining converges quickly to a pattern even with naive scheduling.

In some cases, using an iteration window that is large enough to maximally exploit the available parallelism results in unjustifiably slow convergence. To identify these cases in this experiment, we find

Name	Latency	Description
ALU	1 cycle	integer add/sub and logical
SHIFT	1 cycle	arithmetic and logical shifts
FALU	3 cycles	floating point add/sub and logical
MUL	3 cycles	integer and floating point multiply
DIV	13 cycles	integer and floating point divide
MEM	2 cycles	cache read (cache miss stalls the processor)
	1 cycle	cache write
BRANCH	4 cycles	conditional branch

Table 1: Functional Unit Kind and Latency

it useful to introduce the notion of "cut-off convergence" which constrains the maximum number of iterations scheduled to some fixed amount; the remainder of any paths that have not converged after the cut-off number of iterations are simply scheduled sequentially. We stress that "cut-off convergence" is a creature of our experiment—its purpose is to identify when code explosion is a problem. In practice one should prefer to use scheduling heuristics designed to prevent code explosion; this topic is discussed further in Section 7.2.

7.1 Architectural Models

Two pipelined VLIW architecture models are used for the experiments: one with homogeneous functional units, and one with heterogeneous functional units. Both models assume a single 64-bit wide register file shared by all functional units. With the exception of two "unlimited resource" experiments used to measure threshold performance results, the register file is assumed to have 32 registers.

Operation latencies for both models, given in Table 1, are similar to the Motorola 88110 Superscalar. An instance of the heterogeneous model has 1, 2, 3, or an unlimited number of each of the functional units defined in Table 1. An instance of the homogeneous model has 2, 4, 8, or an unlimited number of homogeneous functional units, where each homogeneous functional unit can perform any of the functions defined in Table 1.

Each VLIW instruction specifies one, possibly NOP, operation for each functional unit. Each operation has the optional side effect of advancing the pipeline. For both models there are no hardware interlocks for detecting data or control hazards, so the compiler is entirely responsible for ensuring that all hazards are avoided at run time.

7.2 Experimental Results

Tables 2 and 3 show the dynamic speedup measured for both target architecture models on 24 Livermore Loops. The speedups are with respect to running the unscheduled code sequentially on the target architecture. Thus, the speedups reflect both the exploitation of multiple functional units and pipelining

kernel	2 FU's (4)	4 FU's (6)	8 FU's (10)	Infx32	InfxInf
LL1	7.3	13.1	17.2	17.2	17.2
LL2	6.3	10.8	14.0	14.0	13.6
LL3	5.8	5.8	5.8	5.8	5.8
LL4	6.9	6.9	6.9	6.9	6.9
LL5	3.7	3.7	3.7	3.7	3.7
LL6	5.4	7.1	7.1	7.1	7.1
LL7	7.6	14.0	16.7	16.6	26.5
LL8	5.7	8.1	9.3	8.0	9.4
LL9	7.7	10.8	11.1	11.1	22.0
LL10	7.8	7.7	7.9	7.8	9.9
LL11	5.5	5.5	5.5	5.5	5.5
LL12	6.9	9.2	9.2	9.2	9.2
LL13	4.9	6.7	6.7	6.7	6.7
LL14	6.5	7.7	7.7	7.6	14.5
LL15	2.0	2.1	2.2	2.2	2.6
LL16	1.8	1.8	1.9	1.9	1.9
LL17	2.3	2.3	2.3	2.3	2.3
LL18	2.4	2.7	3.1	3.1	8.6
LL19	3.1	3.1	3.1	3.1	3.1
LL20	1.9	1.9	1.9	1.9	1.9
LL21	6.1	8.3	8.3	8.3	8.3
LL22	7.2	9.9	11.9	11.9	14.0
LL23	3.9	3.5	4.3	3.9	4.5
LL24	2.8	3.4	3.4	3.4	3.4
Avg	5.1	6.5	7.1	7.1	8.7

Table 2: Homogeneous Multi-cycle Functional Units: SPEED-UP

kernel	1 of each (6)	2 of each (8)	3 of each (10)	Infx32	InfxInf
LL1	9.0	17.3	17.3	17.2	17.2
LL2	7.0	11.7	14.1	14.0	13.6
LL3	5.8	5.8	5.8	5.8	5.8
LL4	7.0	6.9	6.9	6.9	6.9
LL5	3.7	3.7	3.7	3.7	3.7
LL6	5.0	7.1	7.1	7.1	7.1
LL7	13.2	16.1	17.0	16.6	26.5
LL8	7.3	9.0	7.9	8.0	9.4
LL9	9.9	11.2	11.2	11.1	22.0
LL10	4.3	7.0	7.3	7.8	9.9
LL11	5.5	5.5	5.5	5.5	5.5
LL12	5.6	9.2	9.2	9.2	9.2
LL13	3.3	4.5	6.7	6.7	6.7
LL14	5.8	7.5	7.8	7.6	14.5
LL15	1.5	2.1	2.2	2.2	2.6
LL16	1.8	1.9	1.9	1.9	1.9
LL17	2.3	2.3	2.3	2.3	2.3
LL18	3.0	3.4	3.2	3.1	8.6
LL19	3.1	3.1	3.1	3.1	3.1
LL20	1.6	1.9	1.9	1.9	1.9
LL21	5.4	8.3	8.3	8.3	8.3
LL22	9.3	11.2	11.9	11.9	14.0
LL23	4.5	4.4	4.3	3.9	4.5
LL24	2.8	3.4	3.4	3.4	3.4
Avg	5.3	6.9	7.1	7.1	8.7

Table 3: Heterogeneous Multi-cycle Functional Units: SPEED-UP

within a functional unit. The first three columns of Table 2 show the speedups for the homogeneous model assuming 32 registers and 2, 4, and 8 homogeneous functional units, respectively. Table 3 shows the same information for the heterogeneous model configured with 1, 2, and 3 units of each type (again assuming 32 registers). The last two columns of each table show threshold performance levels that are discussed below.

For the results presented in the first 3 columns of each table, loops were pipelined with progressively larger iteration windows until there was no noticeable increase in the average speedup for all benchmarks. The numbers in parentheses at the top of each column show the smallest iteration window sizes for which the highest average performance was attained for each and for which the speedups shown in the tables were generated. Notice that all of the window sizes are fairly small and none exceeds 10 iterations. For the results shown in the first three columns, no more than 64 iterations are scheduled (i.e. this is the cut-off). In almost all cases, only a fraction of this number is needed for convergence to a pattern. The "conv type" column in Tables 4 and 5 how the algorithm terminated: P indicates convergence to a pattern and C indicates cut-off convergence on at least one path.

The last two columns of Tables 2 and 3, which are identical for both tables, show the speedups obtained assuming an unlimited number of functional units and either 32 registers (column "Infx32") or an unlimited number of registers (column "InfxInf"). For both columns we want to show the maximum speedup that can be obtained for the specific architecture configuration given the fixed code motion capabilities, scheduling heuristics, and front-end optimizations used in our system. Therefore, for the "Infx32" and "InfxInf" columns, the iteration window size and cut-off limits were set to the number of iterations that would be executed by each loop at run time, which for most of these loops is 100 iterations. Thus, loops that exhibit natural convergence are guaranteed to be optimal in the sense defined for Theorem 8.2, and the few loops that do not converge are optimal in the same sense because they are fully unrolled and scheduled. Note that by 8 homogeneous functional units and 3 of each heterogeneous functional units, the speedups are already optimal with respect to the "Infx32" numbers, but were obtained with an iteration window size of just 10 iterations. For 16 of the 24 loops, the "8 FU's" and "3 of each" speedups are optimal with respect to the "InfxInf" numbers, which shows that even optimal register allocation for these loops cannot increase performance.

A wide range of speedups exist in Tables 2 and 3, ranging from 1.9 all the way up to 17.2. What we have tried to show is that given a fixed set of code motion capabilities, scheduling heuristics, and front-end optimizations, such as those produced by GCC, our software pipelining algorithm is able to achieve the same performance as fully unrolling and scheduling the loop. Furthermore, despite the generality of our approach, the algorithm manages to achieve good utilization of resources, even with naive scheduling heuristics. The overall performance of these benchmarks, with either pipelining or complete unrolling, could be improved in a number of ways that are orthogonal to our software pipelining approach (e.g. by improving memory reference disambiguation).[4]

[4]Although, due to cyclic dependencies involving long latency operations, such as floating-point division and/or procedure calls, the performance of some of these loops (e.g. LL20), would not likely improve significantly, even with perfect

	2 FU's					4 FU's					8 FU's				
kernel	conv type	reg use	min loop	max loop	total size	conv type	reg use	min loop	max loop	total size	conv type	reg use	min loop	max loop	total size
LL1	P	17	124	124	642	C	22	280	280	2254	P	24	25	25	344
LL2	P	25	100	100	447	P	27	45	45	404	P	28	39	39	411
LL3	P	9	94	94	448	P	11	53	53	437	P	16	77	77	887
LL4	P	14	107	107	634	P	15	45	45	479	P	20	74	74	997
LL5	P	20	44	44	360	P	26	39	39	467	P	32	48	48	713
LL6	P	16	25	25	313	P	18	81	81	1046	P	24	72	72	1295
LL7	P	23	550	550	2831	C	32	681	681	6589	C	32	571	571	5734
LL8	P	28	256	256	1853	C	32	446	446	2798	C	32	399	399	2634
LL9	P	32	148	148	1185	P	32	518	518	4379	P	32	162	162	1449
LL10	P	26	205	205	1964	C	32	119	119	709	C	32	117	117	697
LL11	P	17	46	46	245	P	20	49	49	394	P	28	82	82	907
LL12	P	10	23	23	88	P	10	11	11	76	P	10	11	11	74
LL13	P	32	227	227	1954	P	32	242	242	2369	P	32	225	225	2346
LL14	P	32	64	332	3142	P	32	79	279	6082	P	32	76	237	2623
LL15	P	32	147	156	4623	P	32	167	170	4593	P	32	159	166	4513
LL16	C	32	137	137	1990	C	32	151	151	2009	C	32	151	151	2009
LL17	P	17	38	44	222	P	17	38	42	219	P	17	38	42	219
LL18	P	32	27	202	984	P	32	144	249	1722	P	32	125	215	1706
LL19	P	22	50	74	1076	P	30	47	56	1512	P	32	62	63	2080
LL20	C	29	202	202	2405	C	30	198	198	2380	C	30	197	197	2385
LL21	P	23	127	127	567	P	28	59	59	636	P	32	58	58	730
LL22	P	26	419	419	3840	P	31	301	301	3934	C	32	316	316	4423
LL23	P	32	254	254	1468	P	32	156	156	953	P	32	174	174	1237
LL24	P	13	28	34	139	P	15	25	34	147	P	15	25	34	147
Avg		23	143	164	1392		26	166	179	1941		28	137	148	1690

	Infx32					InfxInf				
kernel	conv type	reg use	min loop	max loop	total size	conv type	reg use	min loop	max loop	total size
LL1	P	24	25	25	344	P	24	25	25	344
LL2	P	32	100	100	824	P	30	20	20	322
LL3	P	16	77	77	887	P	16	77	77	887
LL4	P	20	74	74	997	P	20	74	74	997
LL5	P	32	48	48	713	P	39	57	57	1030
LL6	P	24	71	71	1294	P	24	71	71	1294
LL7	C	32	558	558	5569	C	63	364	364	7708
LL8	C	32	417	417	2684	C	50	391	391	3072
LL9	P	32	108	108	1030	P	88	87	87	2996
LL10	C	32	117	117	690	C	63	110	110	759
LL11	P	28	82	82	907	P	28	82	82	907
LL12	P	10	11	11	74	P	10	11	11	74
LL13	P	32	274	274	2617	P	32	274	274	2617
LL14	P	32	76	194	2282	P	107	23	109	4579
LL15	P	32	162	165	4870	C	51	221	221	5236
LL16	C	32	151	151	2009	C	35	140	140	1953
LL17	P	17	38	42	219	P	17	38	42	219
LL18	P	32	78	193	1357	C	55	749	749	1315
LL19	P	32	62	63	2084	P	42	61	61	2932
LL20	C	30	197	197	2385	C	30	197	197	2385
LL21	P	32	58	58	742	P	38	70	70	1138
LL22	C	32	316	316	4423	P	38	194	194	3963
LL23	P	32	134	134	1171	P	83	79	79	2959
LL24	P	15	25	34	147	P	15	25	34	147
Avg		28	136	146	1680		42	143	147	2076

Table 4: Homogeneous Multi-cycle Functional Units: MISC. PERFORMANCE MEASURES

	1 of each					2 of each					3 of each				
	conv	reg	min	max	total	conv	reg	min	max	total	conv	reg	min	max	total
kernel	type	use	loop	loop	size	type	use	loop	loop	size	type	use	loop	loop	size
LL1	P	16	60	60	308	P	21	38	38	413	P	21	25	25	288
LL2	P	23	42	42	241	P	25	51	51	393	P	27	23	23	276
LL3	P	8	21	21	101	P	12	62	62	520	P	16	77	77	887
LL4	P	10	18	18	132	P	17	59	59	625	P	19	74	74	876
LL5	P	26	76	76	663	P	32	55	55	791	P	32	54	54	795
LL6	P	17	38	38	289	P	20	124	124	1619	P	22	78	78	1378
LL7	P	29	98	98	962	C	32	585	585	6126	C	32	549	549	5599
LL8	P	25	221	221	1501	C	32	404	404	2467	C	32	441	441	2724
LL9	P	32	135	135	2029	P	32	157	157	2155	P	32	178	178	1888
LL10	C	32	238	238	1215	C	32	161	161	999	C	32	136	136	777
LL11	P	8	15	15	78	P	25	65	65	568	P	29	83	83	880
LL12	P	8	26	26	83	P	9	12	12	73	P	9	12	12	74
LL13	P	32	230	230	1880	P	32	210	210	2032	P	32	275	275	2620
LL14	P	32	133	159	1957	P	32	75	107	1576	P	32	75	205	2264
LL15	P	32	29	29	971	C	32	224	224	5765	P	32	161	164	4967
LL16	C	32	147	147	2122	C	32	138	138	1986	C	32	151	151	2006
LL17	P	17	38	40	215	P	17	38	43	216	P	17	38	42	219
LL18	P	32	87	171	1052	P	32	84	228	1527	P	32	89	220	1653
LL19	P	28	91	93	1915	P	32	53	61	1965	P	32	53	63	1957
LL20	P	31	228	228	625	C	30	197	197	2394	C	30	197	197	2391
LL21	P	21	40	40	238	P	32	124	124	1222	P	32	73	73	848
LL22	P	32	298	298	3615	C	32	339	339	4529	C	32	318	318	4433
LL23	P	32	204	204	876	P	32	127	127	1002	P	32	192	192	1383
LL24	P	13	28	37	143	P	14	25	32	134	P	15	25	34	147
Avg		24	106	111	967		27	142	150	1712		27	141	153	1722

	Infx32					InfxInf				
	conv	reg	min	max	total	conv	reg	min	max	total
kernel	type	use	loop	loop	size	type	use	loop	loop	size
LL1	P	24	25	25	344	P	24	25	25	344
LL2	P	32	100	100	824	P	30	20	20	322
LL3	P	16	77	77	887	P	16	77	77	887
LL4	P	20	74	74	997	P	20	74	74	997
LL5	P	32	48	48	713	P	39	57	57	1030
LL6	P	24	71	71	1294	P	24	71	71	1294
LL7	C	32	558	558	5569	C	63	364	364	7708
LL8	C	32	417	417	2684	C	50	391	391	3072
LL9	P	32	108	108	1030	P	88	87	87	2996
LL10	C	32	117	117	690	C	63	110	110	759
LL11	P	28	82	82	907	P	28	82	82	907
LL12	P	10	11	11	74	P	10	11	11	74
LL13	P	32	274	274	2617	P	32	274	274	2617
LL14	P	32	76	194	2282	P	107	23	109	4579
LL15	P	32	162	165	4870	C	51	221	221	5236
LL16	C	32	151	151	2009	C	35	140	140	1953
LL17	P	17	38	42	219	P	17	38	42	219
LL18	P	32	78	193	1357	C	55	749	749	1315
LL19	P	32	62	63	2084	P	42	61	61	2932
LL20	C	30	197	197	2385	C	30	197	197	2385
LL21	P	32	58	58	742	P	38	70	70	1138
LL22	C	32	316	316	4423	P	38	194	194	3963
LL23	P	32	134	134	1171	P	83	79	79	2959
LL24	P	15	25	34	147	P	15	25	34	147
Avg		28	136	146	1680		42	143	147	2076

Table 5: Heterogeneous Multi-cycle Functional Units: MISC. PERFORMANCE MEASURES

In the rest of this section we discuss and interpret the performance results in more detail. To aid in interpreting the results shown in Tables 2 and 3, we present the following performance measures in Tables 4 and 5:

conv type Convergence type: P means that pipelining converged on a pattern and C means that it converged on the cut-off.

reg use The maximum number of registers used at any instruction (i.e. state).

min loop The number of instructions on the shortest path through the pipelined loop.[5]

max loop The number of instructions on the longest path through the pipelined loop.

total size The total number of instructions in the benchmark, including inner loop instructions as well as all code preceding and succeeding the loop.

As discussed in Section 6.1, software pipelining sometimes inserts register-to-register transfers in order to speed convergence of the algorithm. Because these transfers can be eliminated by copy propagation (albeit at the cost of an increase in code size) we have not counted them in the speedup figures in Tables 2 and 3. Even if the copies are not eliminated, the figures in Tables 4 and Table 5 show that the performance penalty is low. For example, for LL2 with 2 homogeneous functional units, the worst case is that all 25 registers are live and must be copied on the backedge, which costs 13 cycles, or 13% of the length of the pipelined loop body. For the large loops the penalty is well below 10%; for small loops the overhead can be reduced by unrolling the pipelined loop body.

There are two interesting anomalies in the speedup tables. The first is that for a few benchmarks, (e.g. LL2, LL8, and LL23 in both tables) the speedup actually decreases slightly after some increases in the number of resources. One cause is that even though two pipelined loops may exhibit the same asymptotic speedup, the overhead from their pre-loop and/or post-loop code can differ (e.g. speedup goes from 14.0 to 13.6 when going from Infx32 to InfxInf for LL2). The other cause of some small decreases in performance when resources increase is overly simplistic scheduling heuristics. For instance, the list scheduling heuristics currently used in the compiler allow operations to be scheduled much earlier than their next use, potentially saturating the register file at subsequent states and thus preventing the removal of false dependencies via renaming that might otherwise allow operations on the critical path to be scheduled earlier. Kernels LL8 and LL23 provide good examples of this effect. The fewer resources there are, the fewer "unimportant" (i.e. off the critical path) operations are scheduled much further head of their next uses and the less likely that the register file becomes saturated with unimportant values. This problem can be alleviated with different scheduling heuristics. In any case, this issue is orthogonal to software pipelining itself.

disambiguation.

[5]Note that since there is only a single path through most of these loops, min loop and max loop are usually equivalent and, in this case, represent the total number of instruction in the pipelined loop.

The other anomaly occurs for loops LL9, LL18, and LL22 in Table 2 and LL7, LL13, and LL18 in Table 3. Factoring out considerations like the above scheduling anomaly and other heuristic aspects such as speculative scheduling, we would expect speedup to increase linearly with the number of functional units until some threshold speedup is reached. Thus, for each doubling in the number of functional units, we would expect the the speedup to be the lesser of twice the old speedup and the maximum (unlimited) speedup; however, for this second class of anomaly, when going from 2 to 4 functional units in Table 2 or 1 to 2 of each functional units in Table 3, we see that the speedup is slightly less than this expected value. The reason for this is that while the iteration window size is chosen to maximize the average speedup shown in the tables, the performance of a few of the loops in each table would have improved with a larger window size, though without any significant effect on the average speedup over all loops.

Finally, it is interesting to consider the circumstances under which the algorithm fails to converge to a pattern before the cutoff is reached. An analysis of the kernels with type "C" convergence (see Tables 4 and 5) shows that the problem arises is vectorizable loops, or, more generally, loops with very few flow dependencies. In this case, the only constraints are resource constraints, and operations are free to move almost anywhere in the schedule. In this situation the lack of dependence structure in the program combined with greedy scheduling heuristics tends to lead to an explosion in the set of states, slowing convergence. Variations on a device of Ebcioğlu's may show how to modify the scheduler to avoid this problem [Ebc87]. The basic idea is to introduce artificial dependencies that don't harm parallelism extraction but dramatically reduce the number of potential states the scheduler may explore. For example, a rule of thumb for vectorizable loops could be that operation x^i must be scheduled no later than the time of x^{i+1}. Since the loop is vectorizable there is no reason to prefer scheduling one before the other; eliminating some orderings reduces the overall number of potential states.

Notice that in most cases, the total size of the final loop is an order of magnitude larger than the shortest and longest paths through the loops. Because loop control conditionals from succeeding iterations are scheduled in parallel with operations from preceding iterations, a new loop exit path is usually created for each iteration scheduled.[6] In some cases this is simply the cost side of a cost vs. performance trade-off inherent to scheduling conditional jumps, the benefits of which are to allow strictly control dependent operations (e.g. operations like stores that can not be renamed) to be scheduled earlier, and to commit to alternative control paths as early as possible so as to minimize the amount speculative scheduling. Fortunately, in many cases, such as for loop exit code, this cost often can be significantly reduced by merging multiple identical control paths into a single shared path. In the context of available operations scheduling, this is accomplished by merging states with identical available operations sets, an optimization we have not implemented.

[6]To guarantee the preservation of correct semantics when scheduling a conditional above operations that precede it, it is necessary to duplicate those operations onto each branch of the conditional after it has been scheduled.

8 On Optimal Software Pipelining

In this section we briefly review research on the limitations of software pipelining, especially a result showing that optimal software pipelining is unachievable [SGE91]. Given this result, we show that our algorithm is "as good as possible" in the sense that it can produce arbitrarily good schedules.

Research in software pipelining has naturally focused on discovering algorithms for computing pipelined schedules, both in general and for specific machines. Concurrently, researchers have investigated the theoretical limitations of software pipelining. One of the central theoretical questions is whether or not there is a software pipelining algorithm that produces optimal pipelined schedules for an arbitrary loop. Because scheduling algorithms are based on preserving data dependences, the natural meaning of "optimality" is with respect to the length of dependence chains.

Definition 8.1 A program L is *time optimal* if for every execution $\langle\langle x_0, s_0\rangle, \ldots, \langle\langle\{\mathit{stop}\}, s_n\rangle\rangle$ of L, n is the length of the longest dependence chain in the execution.

The obvious form of the optimality question is stated as follows: is there an algorithm which takes as input a machine description (i.e., resource constraints, instruction timings, etc.) and a loop, and produces a time optimal schedule for that machine? This problem statement is not very useful, however, because scheduling problems with finite resources are computationally intractable even without software pipelining. To gain some insight into software pipelining itself, researchers have usually abstracted the problem as: given sufficient resources and a loop L, is there an algorithm which computes a time optimal schedule for L?

The answer to this question is trivially "no" for some programs, such as the one in Figure 2. Recall that instructions d and e cannot be scheduled in the same instruction because they write the same store location. One branch of the test must always be optimized at the expense of the other branch, and thus there does not exist a parallel version L that is time optimal. The conflict between d and e in Figure 2 is usually classified as another type of dependence—an *output* dependence [KKP$^+$81]. To avoid this problem, we can rephrase the question again: given unbounded resources and a loop L without output dependences, is there an algorithm which computes a time optimal schedule for L? This question has been resolved negatively [SGE91]. Again the problem is that for some loops an optimal closed-form parallel version does not exist.

While Definition 8.1 is natural, it appears that so many qualifications are required to apply it in the analysis of general software pipelining algorithms that it ceases to be useful. For our purposes we adopt a different definition of what it means for a software pipelining algorithm to be "as good as possible." Recall from Section 4 that the loop L_∞ is the (infinite) parallel program that results from scheduling with complete information about available operations. While L_∞ may not be "optimal", it represents the best that can be done with global knowledge of the program and the ability to fully unroll loops. The following theorem shows that as the window size k of the software pipelining algorithm increases, the quality of the code approaches that of L_∞.

Theorem 8.2 Let L be a loop and let L_k be the result of applying *pipeline* with a scheduling window of k iterations. Let s be any store such that $\mu(L,s) \neq \perp$. Define $t(L,s)$ to be the length of the execution of L in store s. Then

$$\lim_{k \to \infty} t(L_k, s) = t(L_\infty, s)$$

Proof: Let i be the largest index of any iteration in the execution of L on store s. For any $k \geq i$ programs L_k and L_∞ have identical executions on store s. □

Theorem 8.2 is a theoretical result, since in practice the scheduling window k cannot cover more than a few iterations. However, it does show that within the framework of our algorithm it is possible to generate arbitrarily good code, subject to the ability of the scheduler to make good scheduling decisions for finite resources.

9 Related Work

Software pipelining is actually a relatively old idea. Programmers in the microcode community software pipelined code by hand for decades [Kog77]. The first semi-automatic technique for software pipelining was proposed by Charlesworth [Cha81]. For an overview of the history of instruction level parallelism, see [RF93].

Today there are variety of algorithms and frameworks for software pipelining. We describe each and discuss its relationship to our own work. Because of the large amount of work in the area, our discussion of each proposal is necessarily brief.

9.1 Modulo Scheduling

Modulo scheduling is an important software pipelining technique introduced by Rau and Glaeser [RG81] and subsequently used as the basis for numerous other algorithms [Lam87, Jon91, RTS92, Huf93, WMHR93]. Modulo scheduling has been used in compilers for the FPS series [Tou84], the polycyclic machine [RG81], and Cydrome's Cydra [Cyd87].

A basic modulo scheduling algorithm works as follows. 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 or dependency constraints, then the schedule is abandoned and the algorithm either backtracks or tries a larger initiation interval.

Modulo scheduling smoothly integrates the simultaneous treatment of resource constraints and software pipelining. The primary disadvantage of modulo scheduling is that it does not apply directly to loops with conditional tests in the loop body. Two extensions have been proposed to overcome this limitation. Lam introduced *hierarchical reduction* to combine modulo scheduling with complex control flow [Lam87] In hierarchical reduction, the "then" and "else" branches of a conditional 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. Hierarchical reduction suffers from several drawbacks. First, some paths are padded with noops, which may slow execution; second, treating the if-then-else as a single statement necessarily overestimates resource requirements, and third, preserving the control structure of the program restricts possible code motions.

A second proposal for integrating modulo scheduling with conditional tests is to use *if-conversion* [AKPW83] before modulo scheduling and *reverse if-conversion* [WHB92, WMHR93] after modulo scheduling. When a loop is if-converted, the expression of control flow is changed from explicit jumps to guarded operations, where each operation of the original loop is guarded by the predicates of the conditionals that control its execution. In this way, all non-trivial control flow in the loop is replaced by data dependences.

Modulo scheduling with if-conversion appears to improve upon modulo scheduling with hierarchical reduction [WMHR93]. However, if-conversion retains the undesirable features of hierarchical reduction to a considerable degree. First, because control flow is expressed as data dependence, speculative execution of operations (i.e., moving operations above conditionals) is not possible, nor is it possible to reorder conditionals for the same reason. Thus, the possible code motions are restricted.[7] In addition, performing if-conversion greatly hinders the management of limited resources during scheduling. If-conversion schedules all the operations in the original loop body in a single basic block. These operations compete for resources during scheduling, including operations that could never execute simultaneously because they appear on different control paths in the original loop. Thus, straightforward modulo scheduling of if-converted loops overestimates resource requirements.

For the case of loops without control flow and unlimited resources, there is considerable commonality between our algorithm and modulo scheduling. For example, in [AN88a], it was shown that a simplified version of our algorithm produces optimal code for loops without conditionals in the body and for machines with sufficient resources. Despite the differences in conception between the two algorithms, this result was later shown to hold for a small modification of modulo scheduling as well [Jon91].

In short, our algorithm combines software pipelining, resource constraints, and handling of control flow with a flexibility not matched by current modulo scheduling techniques. In our opinion, the significant practical advantage of modulo scheduling at this time is that in cases where both techniques produce equally fast schedules, the schedules produced by modulo scheduling are generally more concise [Jon91].

9.2 Pipeline Scheduling

The work most closely related to our own is that of Ebcioğlu [Ebc87] and Ebcioğlu and Nakatani [NE89, EN90, NE90] and later Moon and Ebcioğlu [ME92]. *Pipeline scheduling* differs from our approach in that the loop body is not constructed by scheduling and testing for repeating states. Instead, the original loop is incrementally transformed to create a parallel schedule. Software pipelining is achieved by moving operations across the backedge of the loop; this has the effect of moving an operation between loop iterations. The handling of control flow is based on the same principles as our own approach and is

[7]This limitation is noted in [WMHR93]; no indication is given about how it can be overcome.

equally general. A scheduling window of operations is also used [NE90], although the purpose is to reduce code explosion rather than to guarantee termination.

An advantage of pipeline scheduling is that the loop is always equivalent to the original loop and therefore it is legal to apply any semantics-preserving transformation to the loop at any time, even transformations that have little to do directly with scheduling. Ebcioğlu and Nakatani exploit this property by aggressively renaming registers and performing strength reduction, optimizations which substantially alter the dependence structure of the loop. We also apply some of these optimizations (see Section 7), but cannot apply them as generally as pipeline scheduling because of our need to guarantee regular dependences for correctness. As an aside, to the best our knowledge, modulo scheduling implementations do not perform any transformations that modify the dependence graph.

An advantage of our algorithm over the current pipeline scheduling algorithm is in the handling of resource constraints. Pipeline scheduling uses only local transformations to move operations from one state to another. Thus, at some points resource constraints may need to be violated in the schedule as an operation moves through one state on its way to another state. To deal with this property, pipeline scheduling has a moderately elaborate phase structure in which resources constraints are alternately enforced and relaxed on specific portions of the loop body. Our algorithm treats resource constraints in a more direct and uniform way.

9.3 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 [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. Our approach and the approach Ebcioğlu and Nakatani is more uniform, overlapping iterations on all paths, rather than a subset of paths.

9.4 Petri Net Techniques

Recently there has been interest in using Petri Nets to formalize the software pipelining problem [GWN91, RA93]. There is a natural mapping from operations, dependences, and resource constraints into Petri Nets, thus combining all of these features in a single, well understood formalism. This approach has been shown to be competitive with modulo scheduling with hierarchical reduction [RA93] and appears promising.

The weakness of current algorithms based on Petri Net techniques is that control flow is handled in a way very similar to if-conversion. The net effect of the mapping into the Petri Net model is that control flow is enforced just like data dependences and thus speculative execution of operations is not possible. Furthermore, the rate execution of iterations is determined by the length of the longest path through the loop body, even when shorter paths through the loop are taken during execution.

10 Conclusions

We have presented a simple but fairly detailed description of a compaction-based software pipelining algorithm that handles resource constraints. The novel aspect of our algorithm is that it cleanly—in fact, completely—separates issues specific to software pipelining, such as detecting repeating "pipeline" states and termination, from other orthogonal issues, such as the computation of available operations and scheduling decisions. We hope that this makes two contributions to the state of the art. First, our algorithm explains in a fairly simple way what software pipelining is about and what are its unique characteristics. Second, the modular and simple design of our algorithm should facilitate the development of general, retargetable implementations of software pipelining.

References

[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.

[Aik88] A. Aiken. *Compaction-Based Parallelization.* PhD thesis, Cornell, 1988. Department of Computer Science Technical Report No. 88-922.

[Aik90] A. Aiken. A theory of compaction-based parallelization. *Theoretical Computer Science*, 73:121–154, 1990.

[AJLS92] V. H. Allan, J. Janardhan, R.M. Lee, and M. Srinivas. Enhanced Region Scheduling on a Program Dependence Graph. In *Proceedings of the 25th International Symposium and Workshop on Microarchitecture (MICRO-25)*, December 1992.

[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.

[AN88a] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–317, June 1988.

[AN88b] 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.

[AN91] A. Aiken and A. Nicolau. A realistic resource-constrained software pipelining algorithm. In *Advances in Languages and Compilers for Parallel Processing*, pages 274–290. MIT Press, 1991.

[Bae80] J. L. Baer. *Computer Systems Architecture*. Computer Press, 1980.

[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.

[Cyd87] Cydrome Inc., Palo Alto, Ca. *Technical Summary*, 1987.

[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.

[EN89] K. Ebcioğlu and A. Nicolau. A global resource-constrained parallelization technique. In *Proceedings of the ACM SIGARCH International Conference on Supercomputing*, June 1989.

[EN90] K. Ebcioğlu and T. Nakatani. A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture. In *Languages and Compilers for Parallel Computing*, pages 213–229. MIT Press, 1990.

[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.

[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.

[GWN91] G. Gao, Y. Wong, and Q. Ning. A timed Petri-Net model for fine grain loop scheduling. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 204–218, June 1991.

[Huf93] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 258–267, June 1993.

[Jon91] R.B. Jones. *Constrained Software Pipelining*. Master's thesis, Department of Computer Science, Utah State University, Logan, UT, September 1991.

[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.

[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.

[Kog77] P. M. Kogge. The microprogramming of pipelined processors. In *Proceedings of the 4th Annual International Symposium on Computer Architecture*, 1977.

[LA92] R.M. Lee and V.H. Allan. Advanced Software Pipelining and the Program Dependence Graph. In *Fourth IEEE Symposium on Parallel and Distributed Processing*, December 1992.

[Lam87] M. Lam. *A Systolic Array Optimizing Compiler.* PhD thesis, Carnegie Mellon University, 1987.

[ME92] S. M. Moon and K. Ebcioğlu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *Proceedings of the 25th International Symposium and Workshop on Microarchitecture (MICRO-25)*, pages 55–71, December 1992.

[NE89] T. Nakatani and K. Ebcioğlu. "Combining" as a compilation technique for VLIW architectures. In *Proceedings of the 22nd Annual Workshop on Microprogramming*, pages 43–55, 1989.

[NE90] T. Nakatani and K. Ebcioğlu. Using a lookahead window in a compaction-based parallelizing compiler. In *Proceedings of the 23rd Annual Workshop on Microprogramming*, 1990.

[Nic85] A. Nicolau. Uniform parallelism exploitation in ordinary programs. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 614–618, August 1985.

[NPA88] A. Nicolau, K. Pingali, and A. Aiken. Fine-grain compilation for pipelined machines. *Journal of Supercomputing*, 1, August 1988.

[PBJ+91] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence flow graphs: An algebraic approach to program dependences. In *Proceedings of the 1991 Symposium on Principles of Programming Languages*, pages 67–78, January 1991.

[PNW92] R. Potasman, A. Nicolau, and H. G. Wang. Register allocation, renaming and their impact on fine-grain parallelism. In *Proceedings of the 1991 Workshop on Languages and Compilers for Parallel Computing*, pages 218–235. Springer Verlag Lecture Notes in Computer Science no. 589, April 1992.

[RA93] M. Rajagopalan and V. H. Allan. Efficient scheduling of fine grain parallelism in loops. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 2–11, December 1993.

[RF93] B. R. Rau and J. Fisher. Instruction-level parallel processing: History, overview, and perspective. *Journal of SuperComputing*, 7(1/2), January 1993.

[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.

[RTS92] B. R. Rau, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 283–299, June 1992.

[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.

[SGE91] U. Schwiegelshohn, F. Gasperoni, and K. Ebcioğlu. On optimal parallelization of arbitrary loops. *Journal of Parallel and Distributed Computing*, 11(2):130–134, 1991.

[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.

[WHB92] N. J. Warter, G. E. Haab, and J. W. Bockhaus. Enhanced Modulo Scheduling for Loops with Conditional Branches. In *Proceedings of the 25th International Symposium and Workshop on Microarchitecture (MICRO-25)*, December 1992.

[WMHR93] N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau. Reverse if-conversion. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 290–299, June 1993.