Software Pipelining

VICKI H. ALLAN

Utah State University

REESE B. JONES

Evans and Sutherland, 650 Komas Drive, Salt Lake City, UT 84108

RANDALL M. LEE

DAKS, 3017 Taylor Avenue, Ogden, UT 84405

STEPHEN J. ALLAN

Utah State Unıversıty

Utilizing parallelism at the instruction level is an important way to improve performance. Because the time spent in loop execution dominates total execution time, a large body of optimizations focuses on decreasing the time to execute each iteration. Software pipelining is a technique that reforms the loop so that a faster execution rate is realızed. Iterations are executed in overlapped fashion to increase parallelism.

Let $\{ABC\}^n$ represent a loop containing operations A, B, C that is executed n times. Although the operations of a single iteration can be parallelized, more parallelism may be achieved if the entire loop is considered rather than a single iteration. The software pipelining transformation utilizes the fact that a loop $\{ABC\}^n$ is equivalent to $A\{BCA\}^{n-1}BC$. Although the operations contained in the loop do not change, the operations are from different iterations of the original loop.

Various algorithms for software pipelinıng exist. A comparison of the alternatıve methods for software pipelining is presented. The relationships between the methods are explored and possibilities for improvement highlighted.

Categories and Subject Descriptors: D.1.3 {**Programming Technique**]: Concurrent Programming; D.3.4 [**Programming Languages**]: Processors—*compilers, optımization*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Instruction level parallelism, loop reconstruction, optimization, software pipelining

Authors' addresses: Vicki H. Allan and Stephen J. Allan, Department of Computer Science at Utah State University, Logan, UT 84322-4205; e-mail: allanv@cs.usu.edu. Reese B. Jones, Evans and Sutherland. Randall M. Lee, DAKS.

CONTENTS

INTRODUCTION

Software pipelining is an excellent method for improving the parallelism in loops even when other methods fail. Emerging architectures often have support for software pipelining.

There are many approaches for improving the execution time of an application program. One approach involves improving the speed of the processor, whereas another, termed *parallel processing*, involves using *multiple processing units*. Often both techniques are used. Parallel processing takes various forms, including processors that are physically distributed, processors that are physically close but asynchronous, and synchronous multiple processors (or multiple functional units). *Fine grain* or *instruction level* parallelism deals with the utilization of synchronous parallelism at the operation level. This paper presents an established technique for parallelizing loops. The variety of approaches to this well understood problem are fascinating. A number of techniques for software pipelining are compared and contrasted as to their ability to deal with necessary complications and their effectiveness in producing quality results. This paper deals not only with practical solutions for real machines, but with stimulating ideas that may enhance current thinking on the problems. By examining both the practical and the impractical approaches, it is hoped that the reader will gain a fuller understanding of the problem and be able to short-circuit new approaches that may be prone to use previously discarded techniques.

With the advent of parallel computers, there are several methods for creating code to take advantage of the power of the parallel machines. Some propose new languages that cause the user to redesign the algorithms to expose parallelism. Such languages may be extensions to existing languages or completely new parallel languages. From a theoretical perspective, forcing the user to redesign the algorithm is a superior choice. However, there will always be a need to take sequential code and parallelize it. Regular loops such as **for** loops lend themselves to parallelization techniques. Many exciting results are available for parallelizing nested loops [Zima and Chapman 1991]. Techniques such as loop distribution, loop interchange, skewing, tiling, loop reversal, and loop bumping are readily available [Wolfe 1990, 1989]. However, when the dependences of a loop do not permit vectorization or simultaneous execution of iterations, other techniques are required. Software pipelining restructures loops so that code from various iterations are overlapped in time.

This type of optimization does not unleash massive amounts of parallelism, but creates a modest amount of parallelism. The utilization of fine grain parallelism is an important topic in machines that have many synchronous functional units. Machines such as horizontal microengines, multiple RISC architectures, VLIW, and LIW can all benefit from the utilization of low level parallelism.

Software pipelining algorithms generally fall into two major categories: modulo scheduling and kernel recognition techniques. This paper compares and contrasts a variety of techniques used to perform the software pipelining optimization. Section 1 introduces terms common to many techniques. Section 2 discusses modulo scheduling; specific instances of modulo scheduling are itemized. Section 2.1 discusses Lam's iterative technique for finding a software pipeline, and in Section 2.2 a mathematical foundation is effectively used to model the problem. In Section 2.3, hardware support for modulo scheduling is discussed. Section 2.4 extends the benefits of predicated execution to machines without hardware support. The next sections discuss the kernel recognition techniques. In Section 3.1, the method of Aiken and Nicolau is presented. Section 3.2 demonstrates the application of Petri nets to the problem, and Section 3.3 demonstrates the construction of an exhaustive technique. Section 4 introduces a completely different technique to accommodate conditionals. The final sections summarize the various contributions of the algorithms and suggest future work.

1. BACKGROUND INFORMATION

1.1 Modeling Resource Usage

Two operations conflict if they require the same resource. For example, if O_1 and O_2 each need the floating point adder (and there is only one floating point adder), the operations cannot execute simultaneously. Any condition that disallows the concurrent execution of two operations can be modeled as a conflict. This is a fairly simple view of resources. A more general view uses the following to categorize resource usage:

(1) *Homogeneous/Heterogeneous.* The resources are termed *homogeneous* if they are identical, hence the operation does not specify which resource is needed, only that it needs one or more resources. Otherwise the resources are termed *heterogeneous.*

(2) *Specific/General.* If resources are heterogeneous and duplicated, we say the resource request is *specific* if the operation requests a specific resource rather than any one of a given class. Otherwise we say the resource request is *general.*

(3) *Persistent/Nonpersistent.* We say a resource request is *persistent* if one or more resources are required after the issue cycle.

(4) *Regular/Irregular.* We say a resource request is *regular* if it is persistent, but the resource use is such that only conflicts at the issue cycle need to be considered. Otherwise we say the request is *irregular.*

A common model of resource usage (heterogeneous, specific, persistent, regular) indicates which resources are required by each operation. The resource reservation table proposed by some researchers models persistent irregular resources [Davidson 1971; Tokoro et al. 1977]. This is illustrated in Figure 1 in which the needed resources for a given operation are modeled as a table in which the rows represent time (relative to instruction issue) and the columns represent resources (adapted from Rau [1994]). For this reservation table, a series of multiplies or adds can proceed one after another, but an add cannot follow a multiply by two cycles because the result bus cannot be shared. This low-level model of resource usage is extremely versatile in modeling a variety of machine conflicts.

			ALU		Multiplier				
Time	Source 1	Source 2	Stage 0	Stage 1	Stage 0	Stage 1	Stage 2	Stage 3	Result Bus
0	X	X							
1			X						
2				X					
3									X

(a)

			ALU		Multiplier				
Time	Source 1	Source 2	Stage 0	Stage 1	Stage 0	Stage 1	Stage 2	Stage 3	Result Bus
0	X	X							
1					X				
2						X			
3							X		
4								X	
5									X

(b)

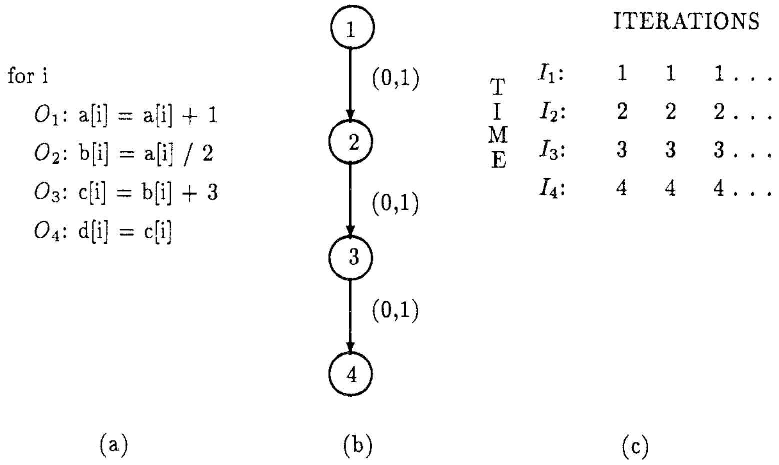
Figure 1. Possible reservation tables for (a) pipelined add and (b) pipelined multiply.

1.2 The Data Dependence Graph

In deciding what operations can execute together, it is important to know which operations must follow other operations. We say a *conflict* exists if two operations cannot execute simultaneously, but it does not matter which one executes first. A *dependence* exists between two operations if interchanging their order changes the results. Dependences between operations constrain what can be done in parallel. A data dependence graph (DDG) is used to illustrate the must-follow relationships between various operations. Let the data dependence graph be represented by $DDG(N, A)$, where N is the set of all nodes (operations) and A is the set of all arcs (dependences). Each directed arc represents a must-follow relationship between the incident nodes. The DDGs for software pipelining algorithms contain true dependences, antidependences, and output dependences. Let O_1 and O_2 be operations such that O_1 precedes O_2 in the original code. O_2 must follow O_1 if any of the following conditions hold: (1) O_2 is *data dependent* on O_1 if O_2 reads data written by O_1, (2) O_2 is *antidependent* on O_1 if O_2 destroys data required by O_1 [Banerjee et al. 1979], or (3) O_2 is *output dependent* on O_1 if O_2 writes to the same variable as does O_1. The term *dependence* refers to data dependences, antidependences, and output dependences.

There is another reason that one operation must wait for another operation. A *control* dependence exists between a and b if the execution of statement a determines whether statement b is executed [Zima and Chapman 1991]. Thus although b is able to execute because all the data is available, it may not execute because it is not known whether it is needed. A statement that executes when it is not supposed to execute could change information used in future computations. Because control dependences have some similarity with data dependences, they are often modeled in the same way [Ferrante et al. 1987].

The dependence information is traditionally represented as a DDG. When there are several copies of an operation (representing the operation in various iterations), several choices present themselves. We can let a different node represent each copy of an operation or we can let one node represent all copies of an operation. We use the latter method. As suspected, this convention does make the graph more complicated to read and requires the arcs be annotated.



```
for i
    $O_1$: a[i] = a[i] + 1
    $O_2$: b[i] = a[i] / 2
    $O_3$: c[i] = b[i] + 3
    $O_4$: d[i] = c[i]
```

	ITERATIONS		
I_1:	1	1	1 . . .
I_2:	2	2	2 . . .
I_3:	3	3	3 . . .
I_4:	4	4	4 . . .

Figure 2. (a) Loop body pseudo-code. (b) Data dependence graph. (c) Schedule.

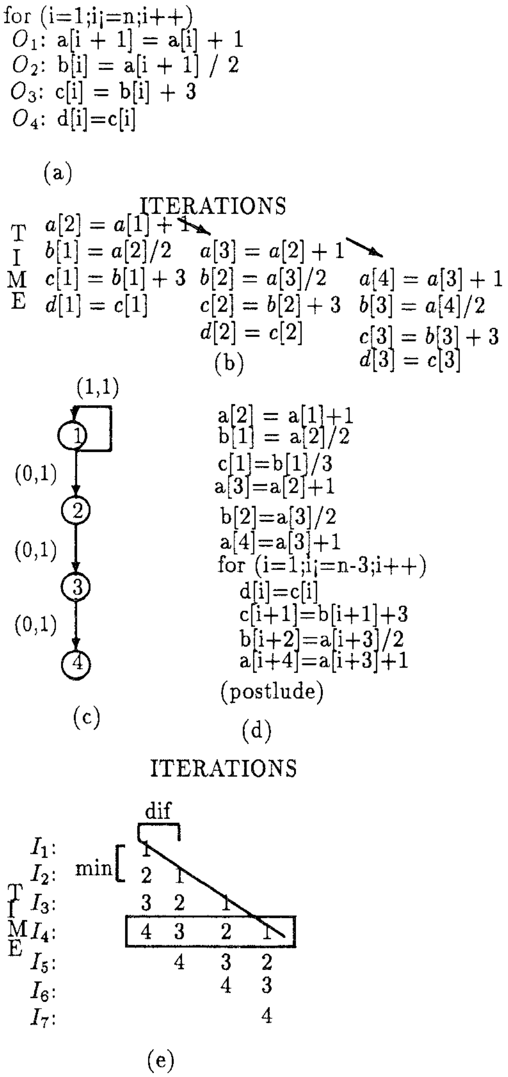
Dependence arcs are categorized as follows. A *loop-independent* arc represents a must-follow relationship among operations of the same iteration. A *loop-carried* arc shows relationships between the operations of different iterations. Loop-carried dependences may turn traditional DDGs into cyclic graphs [Zima and Chapman 1991]. (Obviously, dependences are not cyclic if operations from each iteration are represented distinctly. Cycles are caused due to the representation of operations.)

1.3 Generating a Schedule

Consider the loop body of Figure 2(a).[1] Although each operation of an iteration depends on the previous operation, as shown by the DDG of Figure 2(b), there is no dependence between the various iterations; the dependences are *loop independent*. This type of loop is termed a *doall* loop as each iteration is given an appropriate loop control value and may proceed in parallel [Kuck and Padua 1979; Chandy and Kesselman 1991]. The assignment of operations to a particular time slot is termed a *schedule*. A schedule is a rectangular matrix of sets of operations in which the rows represent time and the columns represent iterations. Figure 2(c) indicates a possible schedule in which all copies of operation 1 can execute together. A set of operations that executes concurrently is termed an *instruction*. All copies of operation 1 execute together in the first instruction. Similarly, all copies of operation 2 execute together in the second instruction. Although it is not required that all iterations proceed in lock step, it is possible if sufficient functional units are present.

Doall loops often represent massive parallelism and hence are relatively easy to schedule. A *doacross* loop is one in which some synchronization is necessary between operations of various iterations [Wolfe 1989]. The loop of Figure 3(a) is an example of such a loop. Operation O_1 of one iteration must precede O_1 of the next iteration because the $a[i]$ used is computed in the previous iteration. Although a doall loop is not possible, some parallelism can be achieved between operations of various iterations. A doacross loop allows parallelism between operations of various loops when proper synchronization is provided.

[1] We use pseudo-code to represent the operations. Even though array accessing is not normally available at the machine operation level, we use this high-level machine code to represent the dependences as it is more readable than RISC code or other appropriate choices. We are not implying our target machine has such operations.



```
for (i=1;i¡=n;i++)
 O1: a[i + 1] = a[i] + 1
 O2: b[i] = a[i + 1] / 2
 O3: c[i] = b[i] + 3
 O4: d[i]=c[i]
```

(a)

```
                ITERATIONS
T  a[2] = a[1] + 1
I  b[1] = a[2]/2     a[3] = a[2] + 1
M  c[1] = b[1] + 3   b[2] = a[3]/2     a[4] = a[3] + 1
E  d[1] = c[1]       c[2] = b[2] + 3   b[3] = a[4]/2
                     d[2] = c[2]       c[3] = b[3] + 3
                                       d[3] = c[3]
```

(b)

(c)

```
a[2] = a[1]+1
b[1] = a[2]/2
c[1]=b[1]/3
a[3]=a[2]+1
b[2]=a[3]/2
a[4]=a[3]+1
for (i=1;i¡=n-3;i++)
   d[i]=c[i]
   c[i+1]=b[i+1]+3
   b[i+2]=a[i+3]/2
   a[i+4]=a[i+3]+1
(postlude)
```

(d)

(e)

Figure 3. (a) Loop body pseudo-code. (b) First three iterations of unrolled loop (c) DDG. (d) Representative code for prelude and new loop body (postlude omitted) pipeline prelude, kernel, and postlude. (e) Execution schedule of iterations. Time (*min*) is vertical displacement. Iteration (*dif*) is horizontal displacement. In this example *min* = 1 and *dif* = 1. The slope (*min/dif*) of the schedule is then 1 which is the length of the kernel.

Because software pipelining enforces the dependences between iterations, but relaxes the need for one iteration to completely finish before another begins, it is a useful fine grain loop optimization technique for architectures that support synchronous parallel execution. The idea behind software pipelining is that the body of a loop can be reformed so that one iteration of the loop can start before previous iterations finish executing, potentially unveiling more parallelism. Numerous systems completely unroll the body of the loop before scheduling to take advantage of parallelism between iterations. Software pipelining achieves an effect similar to unlimited loop unrolling.

Inasmuch as adjacent iterations are overlapped in time, dependences between various operations must be identified. To see the effect of the dependences in Figure 3(a), it is often helpful to unroll a few iterations as in Figure 3(b). Figure 3(c) shows the DDG of the loop body. In this example, all dependences are true dependences. The arcs 1 → 2, 2 → 3, and 3 → 4 are loop independent and the arc 1 → 1 is a loop-carried dependence. The difference (in iteration number) between the source operation and the target operation is denoted as the first value of the pair associated with each arc. Figure 4 shows a similar example in which the loop-carried dependence is between iterations that are two apart. With this less restrictive constraint, the iterations are more overlapped.

It is common to associate a delay with an arc, indicating that a specified number of cycles must elapse between the incident operations. Such delay is used to specify that some operations are multicycle, such as a floating point multiply. An arc $a \rightarrow b$ is annotated with a *min* time that is the time that must elapse between the time the first operation is executed and the time the second operation is executed. Identical operations from separate iterations may be associated with distinct nodes. An alternative representation of nodes lets one node represent the same operation from *all* iterations. Because operations of a loop behave similarly in all iterations, this is a reasonable notation. However, a dependence from node *a* in the first iteration to *b* in the third iteration must be distin-

```
for (i=1;i<=n;i++)
    O1: a[i + 2] = a[i] + 1
    O2: b[i] = a[i + 2] / 2
    O3: c[i] = b[i] + 3
    O4: d[i]=c[i]
```

(a)

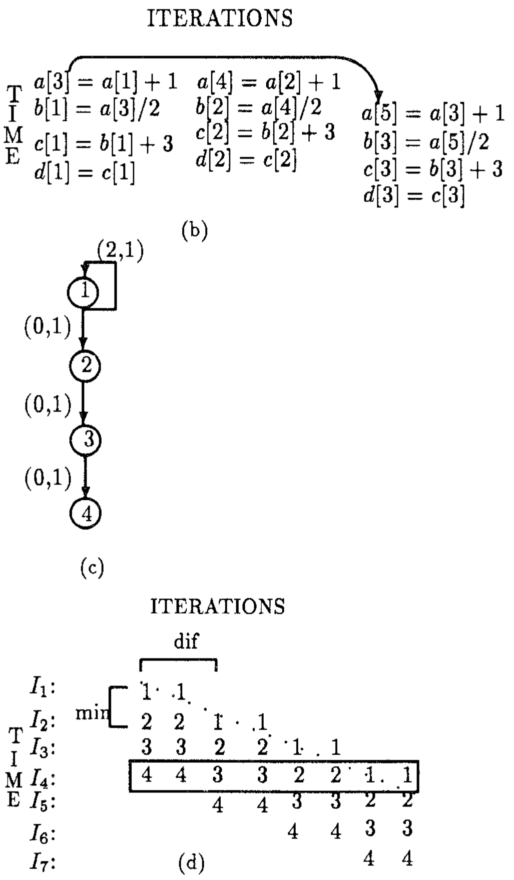


Figure 4. (a) Loop body code. (b) First three iterations of unrolled loop (c) DDG. (d) Execution schedule of iterations.

guished from a dependence between a and b of the same iteration. Thus in addition to being annotated with *min* time, each dependence is annotated with the *dif* that is the difference in the iterations from which the operations come. To characterize the dependence, a dependence arc, $a \rightarrow b$, is annotated with a (dif, min) dependence pair. The *dif* value indicates the number of iterations the dependence spans, termed the *iteration difference*. If we use the convention that a^m is the version of a from iteration m, then $(a \rightarrow b, dif, min)$ indicates there is a dependence between a^m and b^{m+dif}, $\forall m$. For loop-independent arcs, *dif* is zero. The minimum delay intuitively represents the number of instructions that an operation takes to complete. More precisely, for a given value of *min*, if a^m is placed in instruction t (I_t), then b^{m+dif} can be placed no earlier than I_{t+min}.

Table 1 shows examples of code that contain loop-carried dependences. For the code shown, a precedes b in the loop. The loop control variable is i, y is any variable, m is an array, and x is any expression. For example, the first row indicates that m is assigned a value two iterations before it is used. Thus it is a true dependence with a *dif* of 2.

If each iteration of the loop in Figure 3(a) is scheduled without overlap, four instructions are required for each iteration as no two operations can be done in parallel (due to the dependences). However, if we consider operations from several iterations, there is a dramatic improvement.[2] In Figure 3(e) we assume four operations can execute concurrently allowing all four operations (from four different iterations) to execute concurrently in I_4.

When operations from all iterations are scheduled simultaneously, operations are scheduled as early (in execution time) as possible. However, if one had to *store* the code from all iterations of the unrolled loop, it would be prohibitive as there would be many copies of each operation. Thus one seeks to minimize the code needed to represent the improved schedule by locating a repeating pattern in the newly formed schedule. The instructions of a repeating pattern are called the *kernel*, $\mathscr{K}$, of the pipeline. In this paper, we indicate a kernel by enclosing the operations in a box as shown in Figure 3(e).

[2] Scheduling in a parallel environment is sometimes called compaction as the schedule produced is shorter than the sequential version.

Table 1. Dependence Examples

Instruction Label	*Instruction*	*DDG Arc*	*Arc Type*	*Dif*
a b	m[i + 2] = x y = m[i]	a → b	true	2

a
b
	y = m[i + 3]
	m[i] = x
		a → b	anti	3

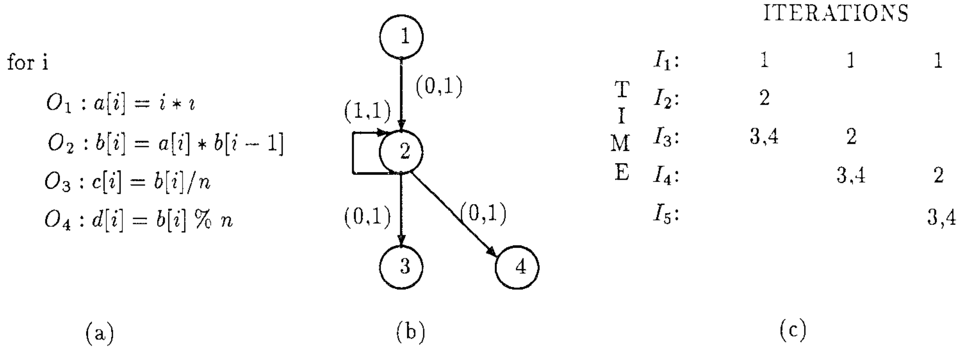
a
b
	m[i] = x
	y = m[i − 2]
		a → b	true	2

a b	y = m[i] m[i − 3] = x	a → b	anti	3

a
b
	y = t
	t = x + i
		a → b
		b → a
			anti
			true
				0
				1

a
b
	t = x + i
	y = t
		a → b
		b → a
			true
			anti
				0
				1

a
b
	y = x + i
	if (x > 2) y = t
		a → b	output	0



```
for i
    $O_1 : a[i] = i * i$
    $O_2 : b[i] = a[i] * b[i - 1]$
    $O_3 : c[i] = b[i]/n$
    $O_4 : d[i] = b[i] \% n$
```

	ITERATIONS		
I_1:	1	1	1
I_2:	2		
I_3:	3,4	2	
I_4:		3,4	2
I_5:			3,4

Figure 5. (a) Loop body code. (b) DDG. (c) Schedule.

The kernel is the loop body of the new loop. Because the work of one iteration is divided into chunks and executed in parallel with the work from other iterations, it is termed a *pipeline*.

There are numerous complications that can arise in pipelining. In Figure 5(a), O_3 and O_4 from the same iteration can be executed together as indicated by 3, 4 in the schedule of Figure 5(c). Note that there is no loop-carried dependence between the various copies of O_1. When scheduling operations from successive iterations as early as dependences allow (termed *greedy* scheduling), as shown in Figure 5(c), O_1 is always scheduled in the first time step I_1. Thus the distance between O_1 and the rest of the operations increases in successive iterations. A cyclic pattern (such as those achievable in other examples) never forms. In the example of Figure 6(b),[3] a pattern does

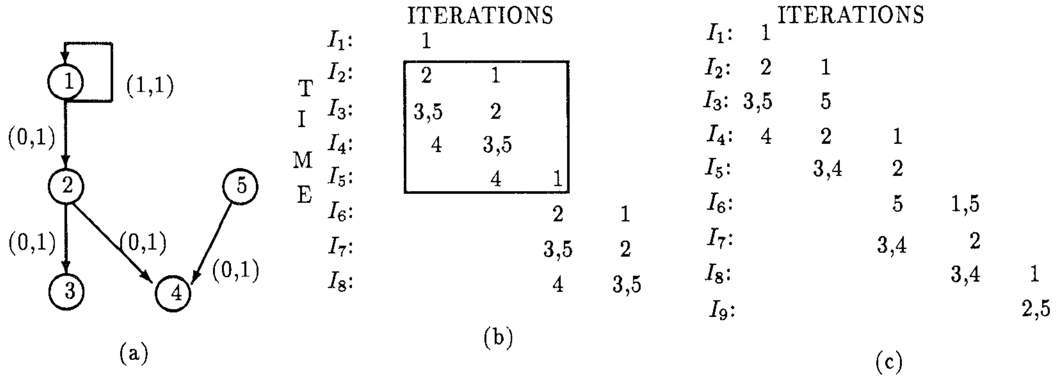


Figure 6. (a) DDG. (b) Schedule that forms a pattern. (c) Schedule that does not form a pattern.

emerge and is shown in the box. Notice it contains *two* copies of every operation. The double-sized loop body is not a serious problem as execution time is not affected, but code size does increase in that the new loop body is four instructions in length. Note that delaying the execution of every operation to once every second cycle would eliminate this problem.

In Figure 6(c) a random (nondeterministic) scheduling algorithm prohibits a pattern from forming quickly. As can be seen, care is required when choosing a scheduling algorithm for use with software pipelining.

1.4 Initiation Interval

In Figure 3(e) a schedule is achieved in which an iteration of the new loop is started in every instruction. The delay between the initiation of iterations of the new loop is called the *initiation interval* (*II*) and is the length of $\mathscr{K}$. This delay is also the *slope* of the schedule that is defined to be *min/dif*. The new loop body must contain all the operations in the original loop. When the new loop body is shortened, execution time is improved. This corresponds to minimizing the effective initiation interval that is the average time one iteration takes to complete. The effective initiation interval is (*II/iteration_ct*), where *iteration_ct* is the number of copies of each operation in the loop $\mathscr{K}$.

Figure 4(d) shows a schedule in which two iterations can be executed in every time cycle (assuming functional units are available to support the eight operations). The slope of this schedule is $min/dif = \frac{1}{2}$. Notice that this slope indicates how many time cycles it takes to perform an iteration. Because two iterations can be executed in one cycle, the effective execution time for one iteration is $\frac{1}{2}$ time cycle.

Notice that because $\mathscr{K}$ does not start or finish in exactly the same manner as the original loop L, instruction sequences α and Ω are required to respectively fill and empty the pipeline. In Figure 3(d), α consists of instructions I_1, I_2, and I_3 and is termed the *prelude*. Ω consists of instructions I_5, I_6, and I_7 and is termed the *postlude*. If the earliest iteration represented in the new loop body is iteration c, and the last iteration represented in the new loop body is iteration d, the *span* of the pipeline is $d - c + 1$. If $\mathscr{K}$ spans n iterations, the prelude must start $n - 1$ iterations preparing for the pipeline to execute, and the postlude must finish $n - 1$ iterations from the point where the pipeline terminates.

[3] It is assumed that a maximum of three operations may be performed simultaneously. General resource constraints are possible, but we assume homogeneous functional units in this example for simplicity of presentation.

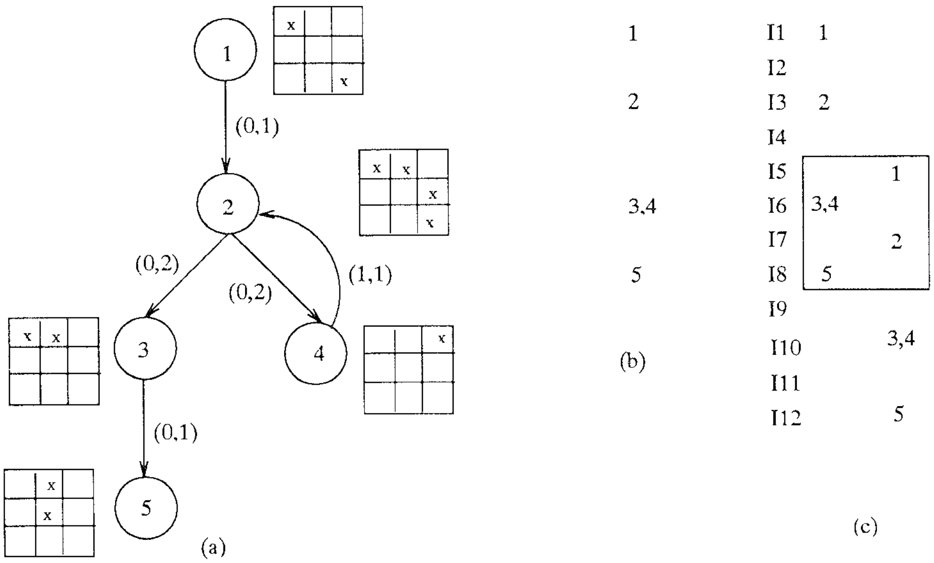


Figure 7. (a) DDG with reservation style resource constraints denoted by boxes. (b) Flat schedule (c) Final schedule, stretched because of resource constraints.

Thus $L^k = \alpha\mathcal{K}^m\Omega$ where k is the number of times L is executed, m is the number of times $\mathcal{K}$ is executed ($m = (k - n + 1)/iteration\_ct$, for $k \geq n$), and α and Ω together execute $n - 1$ copies of each operation.

1.5 Factors Affecting the Initiation Interval

Resource Constrained II. Some methods of software pipelining require an estimate of the initiation interval. The initiation interval is determined by data dependences and the number of conflicts that exist between operations. The resource usage imposes a lower bound on the initiation interval (II_{res}). For each resource, we compute the schedule length necessary to accommodate uses of that resource. For the example of Figure 7(a), O_1 requires resource 1 at cycle 1 (from the time of issue) and resource 3 at cycle 3. If we count all resource requirements for all nodes, it is clear that resource 1 is required 3 times, resource 2 is required 4 times, and resource 3 is required 4 times. Thus at least four cycles are required for the kernel containing all nodes. The relative scheduling of each operation of the original iteration is termed a *flat schedule*, denoted $\mathcal{F}$, and shown in Figure 7(b). The schedule with a kernel size of 4 is shown in Figure 7(c).

Dependence Constrained II. Another factor that contributes to an estimate of the lower bound on the initiation interval is cyclic dependences. There are several approaches for estimating the cycle length due to dependences, II_{dep}. We extend the concept of (dif, min) to a path. Let θ represent a cyclic path from a node to itself. Let min_θ be the sum of the min times on the arcs that constitute the cycle and let dif_θ be the sum of the dif times on the constituent arcs. In Figure 8, we see that the time between the execution of a node and itself (over three iterations, in this case) depends on II. In general, the time that elapses between the execution of a and another copy of a that is dif_θ iterations away is $II * dif_\theta$. II must be large enough so that $II * dif_\theta \geq min_\theta$. Because each iteration is offset dif_θ, after II iterations, $II * dif_\theta$ time steps have passed. Arcs in the DDG fol-

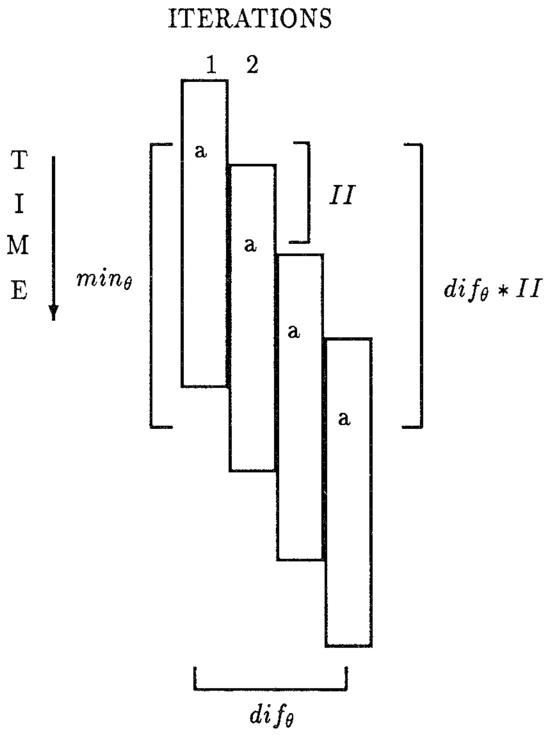


Figure 8. The effect of *II* on cyclic times.

low the transitive law, therefore, a path θ containing a series of arcs with a sum of the minimum delays (min_θ) and a sum of the iteration differences (dif_θ) is functionally equivalent to a single arc from the source node of the path to the destination node with a dependence pair (dif_θ, min_θ). As the cyclic dependences also must be satisfied, the transitive arc $a \to a$, representing a cycle, must satisfy the dependence constraint inequality where the function $\sigma(x)$ returns the sequence number of the instruction in which the operation sequence x begins in $\mathcal{F}$. Let $Time(x^i)$ represent the actual time in which operation x from iteration i is executed. Then the following formula results:

$$\forall\ cycles\ \theta,\ Time(a^{1+dif_\theta}) - Time(a^1) \geq min_\theta .$$

In other words, the time difference in which cyclically dependent operations are scheduled must exceed the minimum time. Because $Time(a^1) = \sigma(a)$ and $Time(a^{1+dif_\theta}) = \sigma(a) + II * dif_\theta$, this formula becomes:

$$\forall\ cycles\ \theta,\ \sigma(a) + II * dif_\theta - \sigma(a) \geq min_\theta .$$

This can be rewritten as:

$$\forall\ cycles\ \theta, 0 \geq min_\theta - II * dif_\theta .$$

The lower bound on the initiation interval due to dependence constraints (II_{dep}) can be found by solving for the minimum value of *II*.

$$\forall\ cycles\ \theta, 0 \geq min_\theta - II_{dep} * dif_\theta \quad (1)$$

$$II_{dep} = \max_{(\forall\ cycles\ \theta)} \left\lceil \frac{min_\theta}{dif_\theta} \right\rceil . \quad (2)$$

For the example of Figure 7, $II_{dep} = 3$ as the only cycle has a *min* of 3 and a *dif* of 1. The actual lower bound on the initiation interval is then $II =$

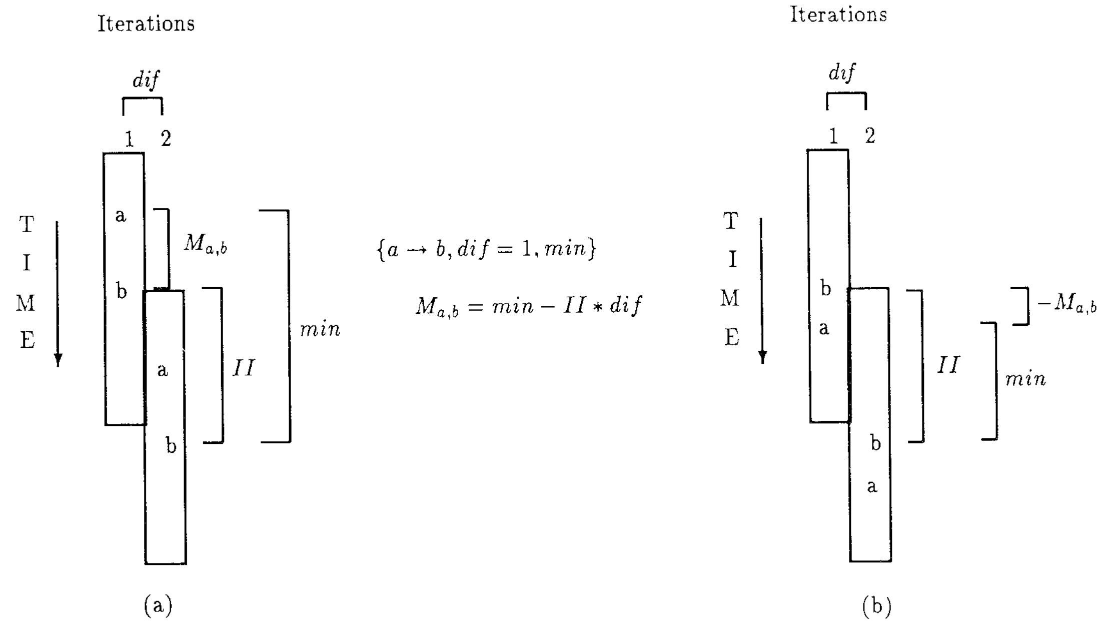


Figure 9. The effect of II on minimum times between nodes. Each rectangle represents the schedule of one iteration of the original loop. (a) Positive value for $M_{a,b}$ indicates a precedes b. (b) A negative value for $M_{a,b}$ indicates a follows b.

$\max(II_{dep}, II_{res})$, which is 4 for this example. Any cycle having min/dif equal to II is termed a *critical cycle.*

1.6 Methods of Computing *II*

1.6.1 Enumeration of Cycles

One method of estimating II_{dep} simply enumerates all the simple cycles [Mateti and Deo 1976; Tiernan 1970]. The maximum (min/dif) for all cycles is then the II_{dep} [Dehnert et al. 1989].

1.6.2 Shortest Path Algorithm

Another method for estimating II_{dep} uses transitive closure of a graph. The transitive closure of a graph is a reachability relationship. If the dependence constraints are expressed as a function of II, a single calculation to compute the transitive closure is sufficient. This symbolic computation of closure allows the closure of the dependence constraints to be calculated independently of a particular value for II. The dependence constraint between nodes in the closure is represented by the set of distances[4] by which the two nodes must be separated in $\mathscr{F}$ in order to satisfy dependences. In the flat schedule, the distance, computed from a single (dif, min) dependence pair for an arc $a \rightarrow b$, is given by $M_{a,b} = min - II * dif$. We would like to compute the minimum distance two nodes must be separated, but as this information is dependent on II (which is a variable) one cannot simply take the maximum distance for all paths between a and b. As shown in Figure 9, we see that an arc from a to b with a dif of 1 implies that b must follow a (in the flat schedule) by $min - II$ time units. In general, an arc $(n_i \rightarrow n_j, dif, min)$ is equivalent to the arc $(n_i \rightarrow n_j, 0, min - dif * II)$, provided II is known. As shown in Figure 9(b), this new minimum value can be negative. For example, $M_{i,j} = -3$ indicates

[4] This distance is referred to as the *cost* in transitive closure algorithms

that n_j can precede n_i in the flat schedule by 3 time units. This computation gives the earliest time n_j can be placed with respect to n_i.

If there are two paths from a to b, one having $(dif, min) = (3, 8)$ and the other having $(dif, min) = (1, 5)$, it is not evident which represents the stricter constraint. In the first case, we have $M_{i,j} = 8 - 3 * II$ whereas in the second case we have $M_{i,j} = 5 - 1 * II$. If $II < 2$, the first is larger. For example, if $II = 1$, $8 - 3 > 5 - 1$. If $II \geq 2$, the (dif, min) for the second arc represents the larger distance. For example, when $II = 2$, $8 - 3 * 2 < 5 - 2$. Thus *both* distances between a and b must be considered unless one can be eliminated by discovering II is sufficiently large. In the previous example, if II is known to be at least 2 (due to the computation of lower bounds), then whenever (1, 5) is satisfied so is (3, 8), and the latter constraint can be ignored. Computing the closure of the dependence constraints is equivalent to finding the longest path between each pair of nodes in the strongly connected component. By Equation (1), we see cycles always have a zero or negative distance. Therefore, by reversing the sense of all inequalities, one can use Floyd's All-Points Shortest Path Algorithm to calculate the all-points longest path [Smith 1987].[5]

Let N be the set of nodes in the graph. The *cost* matrix C is defined as follows: entry C_{ij} contains the set of all dependence information that influences the longest path between i and j, that is, the (dif, min) dependence pairs representing the transitive closure of the dependence arcs between node i and node j. Initially the C_{ij} set contains the (dif, min) dependence pairs of arcs in the strongly connected component. The closure of the dependence constraint is calculated as follows:

$$\forall k \in N \, \forall i \in N \, \forall j \in N$$

$$C_{ij} = MaxCost\big(C_{ij}, AddCost(C_{ik}, C_{kj})\big).$$

The function *AddCost* creates an updated cost set by adding every dependence pair in the first set to every dependence pair in the second set, component-wise. The function *MaxCost* returns a set that is the union of the two sets with the redundant dependence pairs removed. A cost is redundant if it is unnecessary because it provides no additional constraints. Determining whether one of (dif_1, min_1) or (dif_2, min_2) is redundant involves two separate tests. If $min_1 - II * (dif_1) \leq min_2 - II * (dif_2)$, then the distance associated with (dif_2, min_2) is currently longer than the distance associated with (dif_1, min_1). If $dif_1 \geq dif_2$, then for all larger values of II, the distance of (dif_2, min_2) remains longer due to the fact that II has a negative coefficient in the distance formula.

Formally, given a dependence pair (dif_1, min_1) and any other dependence pair (dif_2, min_2) in the distance set, then (dif_1, min_1) is *redundant* if $dif_1 \geq dif_2$ and $min_1 - min_2 \leq II * (dif_1 - dif_2)$. From the previous example in which $\text{pair}_1 = (3, 8)$ and $\text{pair}_2 = (1, 5)$, if we assume $II = 2$, pair_1 can be shown to be redundant as $3 > 1$ and $8 - 5 \leq 2 * (3 - 1)$.

For an initiation of II, the cost function ($cost^{II}(i, j)$) gives the number of instructions by which node j must follow node i in the flat schedule. The dependence constraint is defined as follows:

$$cost^{II}(i, j) = \max_{(\forall (dif, min) \in C_{ij})} min - II * dif.$$

After the closure of the dependence constraints is calculated, the dif_θ and min_θ values for all cycles θ in the DDG are available to calculate II_{dep} (the minimum initiation interval due to dependence constraints). This is similar to computing the all-points longest path of

[5] Floyd's original algorithm handles negative costs if all cycles have positive costs.

Table 2. Transitive Closure of Graph

Source Node	*Destination Node*				
	1	2	3	4	5
1	(2, 6),(3,6)	(1, 4),(2,4)	(0, 2),(1,2)	(0, 3),(1,3)	(0, 1)
2	(1, 2)	(1, 4),(2,6),(3,6)	(0, 2),(1,4),(2,4)	(0,3),(1,5),(2,5)	(1, 3)
3	(2, 4)	(1, 2)	(1, 4),(2,6),(3,6)	(0, 1)	(2, 5)
4	(2, 3)	(1, 1)	(1, 3),(2,5),(3,5)	(1, 4),(2,6),(3,6)	(2, 4)
5	(3, 5)	(2, 3)	(1, 1)	(1, 2)	(3, 6)

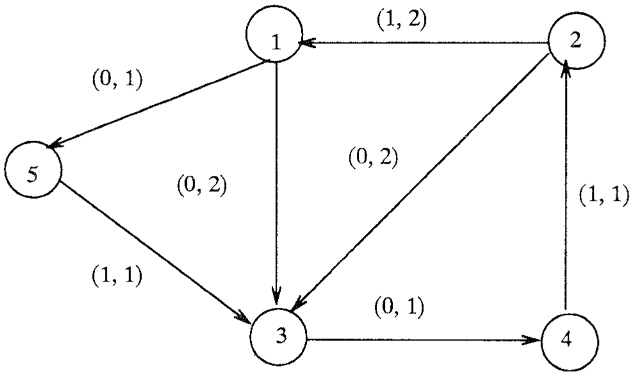


Figure 10. Sample graph.

a graph, the only difference being that the distance depends on II. The dif_θ and min_θ values for a cycle θ containing node i are found in the C_{ii} cost set, for example, the entries of the main diagonal.

Table 2 shows C, the result of calculating the closure of the dependence constraints for the graph of Figure 10. Every simple path between nodes is represented in this closure table. Trying a few simple cases will convince you that complex cycles (having more repeated nodes than the initial and final node) only add redundant information when computing the maximum (dif, min) of a cycle. However, no attempt has been made to throw out other redundant information in this example. Once the closure of the dependence constraints has been calculated, a lower bound on the initiation interval due to dependence constraints can be found using Equation (2). The (dif, min) pairs on the diagonal of Table 2 are the dependence pairs corresponding to the cycles in the graph. Substituting the dependence pairs from the diagonal of Table 2 into Equation (2) yields:

$$II_{dep} = \max\left(\left\lceil \frac{6}{2} \right\rceil, \left\lceil \frac{4}{1} \right\rceil, \left\lceil \frac{6}{3} \right\rceil\right) = 4.$$

Four is the estimate on the lower bound of the initiation interval due to dependence constraints.

1.6.3 Iterative Shortest Path

The method for computing II_{dep} can be simplified if one is willing to recompute the transitive closure for each possible II. For a given II, it is clear which of two (dif, min) pairs is more restrictive. Thus the processing is much simpler as the cost table need only contain one (dif, min) pair [Huff 1993; Zaky 1989]. Because the cost of computing transitive closure grows as the square of the num-

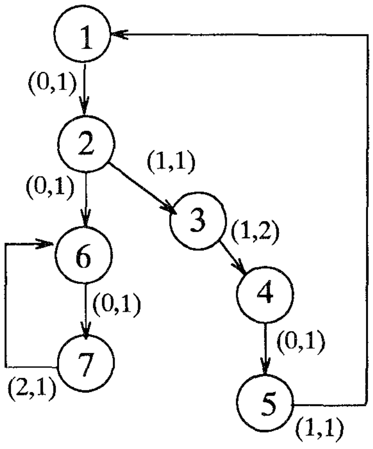


Figure 11. Sample graph.

ber of values at a cost entry, this is a sizable savings.

Path algebra is an attempt to formulate the software pipelining problem in rigorous mathematical terms [Zaky 1989]. Zaky constructs a matrix M that indicates for each entry $M_{i,j}$ the *min* time between the nodes i and j. This construction is simple in the event that the *dif* value between two nodes is zero. Assume $a = n_i$ and $b = n_j$. If there is an arc $(a \to b,\ dif = 0,\ min)$, $M_{i,j} = min$. As is shown in Figure 9, we see that an arc $(a \to b,\ dif = 1,\ min)$ implies than n_j must follow n_i by $min - II$ time units. In general, an arc $(n_i \to n_j,\ dif,\ min)$ represents the distance $min - dif * II$. This computation gives the *earliest* time n_j can be placed with respect to n_i in the flat schedule. The drawback is that before we are able to construct this matrix, we must estimate II. The technique allows us to tell if the estimate for II is large enough and iteratively try larger II until an appropriate II is found.

	1	2	3	4	5	6	7
1	$-\infty$	1	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
2	$-\infty$	$-\infty$	-1	$-\infty$	$-\infty$	1	$-\infty$
3	$-\infty$	$-\infty$	$-\infty$	0	$-\infty$	$-\infty$	$-\infty$
4	$-\infty$	$-\infty$	$-\infty$	$-\infty$	1	$-\infty$	$-\infty$
5	-1	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
6	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	1
7	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-3	$-\infty$

(a) Original Matrix M

	1	2	3	4	5	6	7
1	$-\infty$	$-\infty$	0	$-\infty$	$-\infty$	2	$-\infty$
2	$-\infty$	$-\infty$	$-\infty$	-1	$-\infty$	$-\infty$	2
3	$-\infty$	$-\infty$	$-\infty$	$-\infty$	1	$-\infty$	$-\infty$
4	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
5	$-\infty$	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
6	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-2	$-\infty$
7	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-2

(b) M^2

	1	2	3	4	5	6	7
1	0	1	0	0	1	2	3
2	-1	0	-1	-1	0	1	2
3	0	1	0	0	1	2	3
4	0	1	0	0	1	2	3
5	-1	0	-1	-1	0	1	2
6	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-2	1
7	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-3	-2

(c) Closure

Figure 12. Closure computation.

Consider the graph of Figure 11. For an estimate of $II = 2$, the matrix M is shown in Figure 12(a). Note that according to this matrix (for restrictions due to paths of length one), n_3 and n_4 can execute together ($M_{3,4} = 0$). Even though there must be a *min* time of 2 between n_3 of one iteration and n_4 from the next as given by $(n_3 \to n_4, 1, 2)$, the delay between iterations (II) is two. Hence no further distance between n_3 and n_4 is required in the flat schedule.

Zaky defines a type of matrix multiply operation, termed *path composition*, such that $M^2 = M \otimes M$ represents the minimum time difference between nodes that is required to satisfy paths of length two. For two vectors (a_1, a_2, a_3, a_4) and (b_1, b_2, b_3, b_4), $(a_1, a_2, a_3, a_4) \otimes (b_1, b_2, b_3, b_4) = a_1 \otimes b_1 \oplus a_2 \otimes b_2 \oplus a_3 \otimes b_3 \oplus a_4 \otimes b_4$. Notice that this is similar to an inner product. $\otimes$ has precedence over $\oplus$. $\otimes$ is addition, and $\oplus$ is maximum.[6] For example, to get $M^2(1, 6)$ we compose row

[6] It may seem strange that $\otimes$ is addition, but the notation was chosen to show the similarity between path composition and inner product.

1 of the preceding matrix with column 6 as follows:

$$\begin{aligned}&[-\infty, 1, -\infty, -\infty, -\infty, -\infty, -\infty]\\ &\quad \otimes [-\infty, 1, -\infty, -\infty, -\infty, -\infty, -3]\\ &\quad = max(-\infty, 2, -\infty, -\infty, -\infty, -\infty, -\infty)\\ &\quad = 2.\end{aligned}$$

Thus there is a path composed of two edges (indicated by the superscript on M) between n_1 and n_6 such that n_6 must follow n_1 by two time steps. We can verify this result by noting that the path which requires n_6 to follow n_1 by two is the path 1 → 2 → 6 that has a (dif, min) of (0, 2).

The matrix M is a representation of the graph in which all *dif* values have been converted to zero. Therefore, edges of the transitive closure are formed from adding the *min* times of the edges that compose the path. Path composition, as just defined, adds transitive closure edges. In the technique of Section 1.6.2, edges of the transitive closure are added by summing both the *dif* and *min* values composing the path. Zaky does the same thing by simply adding the minimum values on each arc of the path. This is identical as *dif*s in Zaky's method are always zero. Because there can be multiple paths between the same two nodes, we must store the maximum distance between the two nodes. Thus the matrix M^2 is shown in Figure 12(b). Formally, we perform regular matrix multiplication, but replace the operations $(*, +)$ with $(\otimes, \oplus)$, where $\otimes$ indicates the *min* times that must be added, and $\oplus$ indicates the need to retain the largest time difference required.

Clearly, we need to consider constraints on placement dictated by paths of *all* lengths. Let the closure of M be $\Gamma(M) = M \oplus M^2 \oplus M^3 \oplus \cdots \oplus M^{n-1}$ where n is the number of nodes and M^i indicates i copies of M path multiplied together. Only paths of length $n - 1$ need to be considered as paths that are composed of more arcs must contain cycles and give no additional information. We propose using a variant of Floyd's algorithm as shown in Figure 13 to make closure more efficient. $\Gamma(M)$ represents the maximum distance between each pair of nodes after considering paths of all lengths.

```
for (k = 0; k<nodect; k++)
    for (i = 0; i<nodect; i++)
        if (M[i][k] > -∞)
            for (j = 0; j<nodect; j++)
            { t =M[i][k]+M[k][j];
                if (t > M[i][j])
                    M[i][j]=t;
            }
```

Figure 13. A variant of Floyd's algorithm for path closure

A legal II will produce a closure matrix in which entries on the main diagonal are nonpositive. For this example, an II of 2 is clearly minimal because of II_{dep}. The closure matrix contains nonpositive entries on the diagonal, indicating an II of 2 is sufficient. If an II of 1 is used, the matrix of Figure 14 results. The positive values on the diagonal indicate II is too small.

Suppose we repeat the example with $II = 3$ as shown in Figure 15. All diagonals are negative in the closure table. For instance, O_1 must follow O_1 by at least -4 time units. In other words, O_1 can *precede* O_1 from the next iteration by 4 time units. Inasmuch as all values along the diagonal are nonpositive, $II = 3$ is adequate. Methods that use an iterative technique to find an adequate II try various values for II in increasing order until an appropriate value is found.

1.6.4 Linear Programming

Yet another method for computing II_{dep} is to use linear programming to minimize II given the restrictions imposed by the (dif, min) pairs [Govindarajan et al. 1994].

1.7 Unrolling / Replication

The term unrolling has been used by various researchers to mean different

	1	2	3	4	5	6	7
1	$-\infty$	1	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
2	$-\infty$	$-\infty$	0	$-\infty$	$-\infty$	1	$-\infty$
3	$-\infty$	$-\infty$	$-\infty$	1	$-\infty$	$-\infty$	$-\infty$
4	$-\infty$	$-\infty$	$-\infty$	$-\infty$	1	$-\infty$	$-\infty$
5	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
6	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	1
7	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	−1	$-\infty$

(a)

	1	2	3	4	5	6	7
1	3	4	4	5	6	8	9
2	2	3	3	4	5	7	8
3	2	3	3	4	5	7	8
4	1	2	2	3	4	6	7
5	3	4	4	5	6	8	9
6	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	0	1
7	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	−1	0

(b)

Figure 14. (a) Original matrix (b) Closure for $II = 1$.

	1	2	3	4	5	6	7
1	$-\infty$	1	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
2	$-\infty$	$-\infty$	−2	$-\infty$	$-\infty$	1	$-\infty$
3	$-\infty$	$-\infty$	$-\infty$	0	$-\infty$	$-\infty$	$-\infty$
4	$-\infty$	$-\infty$	$-\infty$	$-\infty$	−1	$-\infty$	$-\infty$
5	−2	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
6	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	1
7	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	−5	$-\infty$

(a)

	1	2	3	4	5	6	7
1	−4	1	−1	−1	−2	2	3
2	−5	−4	−2	−2	−3	1	2
3	−3	−2	−4	0	−1	−1	0
4	−3	−2	−4	−4	−1	−1	0
5	−2	−1	−3	−3	−4	0	1
6	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	−4	1
7	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	−5	−4

(b)

Figure 15. (a) Original matrix. (b) Closure for $II = 3$.

transformations. A loop is *completely unrolled* if all iterations are concatenated as in Figure 16(b). We use the term *replicated* when the body of the loop is copied a number of times and the loop count adjusted as in Figure 16(c). Often the term *unrolling* is used to represent either concept [Rau et al. 1992; Zima and Chapman 1991]. We use the term unrolling to represent complete unrolling and replication to represent making (fewer than the loop count) copies of the loop body. All replicated copies must exist in the newly formed schedule.

Replication is helpful in two different ways. For algorithms in which iteration differences of greater than one cannot be handled, replication eliminates the occurrence of these nonunit iteration differences. For other algorithms, replication allows fractional initiation intervals by letting adjacent iterations be scheduled differently. Time optimality is possible because the new loop body can include more than one copy of each operation. This is an advantage that can be achieved by any technique by simple replication, but is complicated by the facts that (1) any replication increases complexity, and (2) it is not known how much replication is helpful. Unrolling is used in order to *find* a schedule (see Section 3) in many methods. Many iterations may be examined to find a naturally occurring loop, but it is not required that there is more than one copy of each operation in the new loop body.

1.8 Support for Software Pipelining

Software pipelining algorithms sometimes require that the loop limit be a run-time constant. Thus the pipeline can be stopped before it starts to execute operations from an iteration that should not be executed. *Speculative execution* refers to the execution of operations before it is clear that they should be executed. For example, consider the loop of Figure 17(a) that is controlled by

```
For i=1 to 4
  a
  b
```

(a)

```
a
b
a'
b'
a"
b"
a'''
b'''
```

(b)

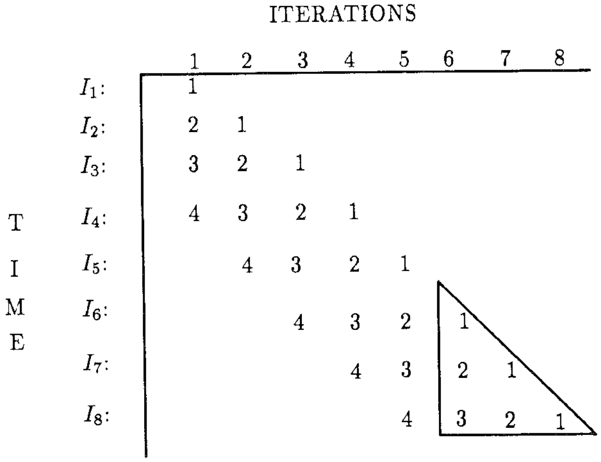
```
For i = 1 to 2
  a
  b
  a'
  b'
```

(c)

Figure 16. (a) Loop code (b) Completely unrolled loop. (c) Replicated loop.

```
for (i = 0; d[i]<MAX;i++)
  O1: a[i + 1] = a[i] + 1
  O2: b[i] = a[i + 1] / 2
  O3: c[i] = b[i] + 3
  O4: d[i] = c[i]
```

(a)



TIME / ITERATIONS	1	2	3	4	5	6	7	8
I_1:	1							
I_2:	2	1						
I_3:	3	2	1					
I_4:	4	3	2	1				
I_5:		4	3	2	1			
I_6:			4	3	2	1		
I_7:				4	3	2	1	
I_8:					4	3	2	1

(b)

Figure 17. (a) Loop body code. (b) Schedule (part enclosed in triangle should not have been executed)

for (i=0; d[i]<MAX; i++). Suppose that 5 iterations execute before d[i] is greater than MAX. The operations in the triangle in Figure 17(b) *should not have been executed.* Because we are executing operations from several iterations, when the condition becomes false, we have executed several operations that would not be executed in the original loop. Because these operations change variables, there must be some facility for "backing out" of the computations. When software pipelining is applied to general loops, the parallelism is not impressive unless there is support for speculative execution. Such speculative execution is supported by various mechanisms, including variable renaming or delaying speculative stores until the loop condition has been evaluated.

2. MODULO SCHEDULING

Historically, early software pipelining attempts consisted of scheduling opera-

F_1:	1
F_2:	
F_3:	2,3
F_4:	4,5
F_5:	6
F_6:	7

(a)

I_1:	1		
I_2:			
I_3:	2,3	1	
I_4:	4,5		
I_5:	6	2,3	1
I_6:	7	4,5	
I_7:		6	2,3
I_8:		7	4,5
I_9:			6
I_{10}:			7

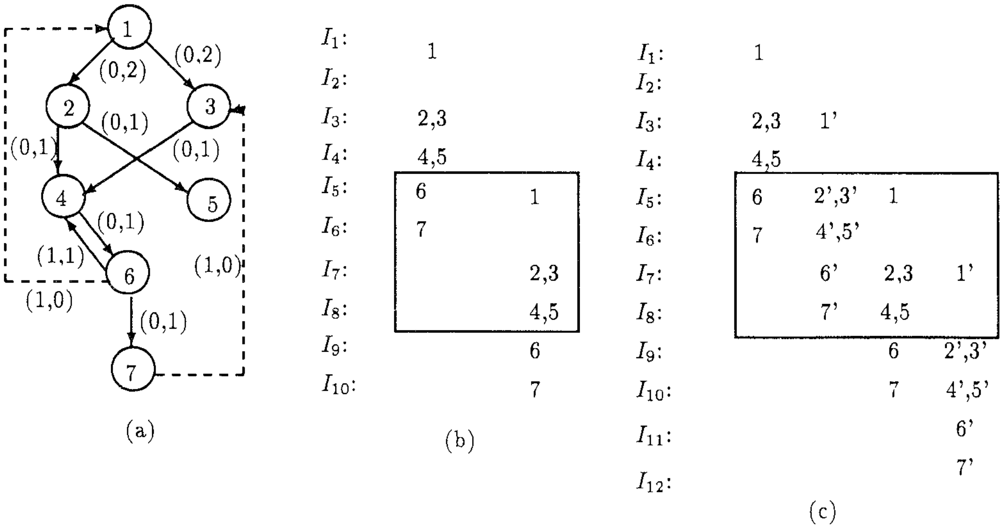
(b)

Figure 18. (a) Flat schedule $\mathcal{F}$ with $II = 2$. (b) The resulting regular pipeline.

tions from several iterations together and looking for a pattern to develop. Modulo scheduling uses a different approach in that operation placement is done so that the schedule is legal in a cyclic interpretation [Rau and Glaeser 1981; Rau et al. 1982]. In other words, when operation a is placed at a given location, one must ensure that if the schedule is overlapped with other iterations, there are no resource conflicts or data dependence violations. In considering the software pipeline of Figure 18(b), a schedule for one iteration (shown in Figure 18(a)) is offset and repeated in successive iterations. If the schedule for one iteration is of length f, there are $\lceil f/II \rceil$ different iterations represented in the kernel (new loop body). For this example, the span is 3 ($\lceil 6/2 \rceil$) as operations in the kernel come from three different iterations. The difficulty is in making sure the placement of operations is legal given that successive iterations are scheduled identically. In making that determination, it is clear that the offset (which is just the initiation interval) is known before scheduling begins. Because of the complications due to resource conflicts, we can only guess at an achievable initiation interval. As the problem is a difficult one, there is no polynomial time algorithm for determining an optimal initiation interval. The problem has been shown to be NP-complete [Hsu and Davidson 1986; Lam 1987]. This problem is solved by estimating II and then repeating the algorithm with increasing values for II until a solution is found.

Locations in the flat schedule (the relative schedule for the original iteration) are denoted $F_1, F_2, \dots, F_f$. The pipelined loop, $\mathcal{K}$, is formed by overlapping copies of $\mathcal{F}$ that are offset by II. Figure 18(a) illustrates a flat schedule and Figure 18(b) shows successive iterations offset by the initiation interval to form a pipelined loop body of length two. This is termed modulo scheduling in that all operations from locations in the flat schedule that have the same value modulo II are executed simultaneously. In this case, operations from F_1, F_3, and F_5 ($F_i : i \bmod 2 = 1$) execute together and F_2, F_4, and F_6 ($F_i : i \bmod 2 = 0$) execute together. This type of pipeline is called a *regular pipeline* in that each iteration of the loop is scheduled identically, that is, $\mathcal{F}$ is created so that if a new iteration is started every II instructions, there are no resource conflicts and all of the dependences are satisfied.

Most scheduling algorithms use list scheduling in which some priority is used to select which of the ready operations is scheduled next. Scheduling is normally as early as possible in the schedule,



(b)

I_1:	1	
I_2:		
I_3:	2,3	
I_4:	4,5	
I_5:	6	1
I_6:	7	
I_7:		2,3
I_8:		4,5
I_9:		6
I_{10}:		7

(c)

I_1:	1			
I_2:				
I_3:	2,3	1'		
I_4:	4,5			
I_5:	6	2',3'	1	
I_6:	7	4',5'		
I_7:		6'	2,3	1'
I_8:		7'	4,5	
I_9:			6	2',3'
I_{10}:			7	4',5'
I_{11}:				6'
I_{12}:				7'

Figure 19. (a) DDG (b) Schedule (c) Schedule after renaming to eliminate loop-carried antidependence.

though some algorithms have tried scheduling as late as possible or alternating between early and late placement [Huff 1993]. In modulo scheduling, operations are placed one at a time. Operations are prioritized by difficulty of placement (a function of the number of legal locations for an operation). Operations that are more difficult to place are scheduled first to increase the likelihood of success. Conceptually, when you place operation a into a partially filled flat schedule, you think of the partial schedule as being repeated at the specified offset. (There are *span* copies of the schedule.) A legal location for a must not violate dependences between previously placed operations in any of these copies and a. In addition, there must not be resource conflicts between operations that execute simultaneously in this schedule.

Consider the example of Figure 19 in which the dependence graph governing the placement is shown along with the schedule. Suppose operation 6 is the last operation to be placed. We determine a range of locations in the flat schedule in which 6 can be placed. Clearly operation 6 cannot be placed earlier than F_5 (I_5 and I_9) as it must follow operation 4 that is located in F_4 (I_4 and I_8). However, this is also the latest it can be scheduled. When we consider iteration 2 (that is offset by the initiation interval of 4), operation 1 from iteration 2 (scheduled in I_5) must not precede operation 6 from iteration 1. Thus there is only one legal location for operation 6 (assuming all other operations have been scheduled). All loop-carried dependences and conflicts between operations are considered as the schedule is built. The newly placed operation must be legal in the series of offset schedules represented by the previously placed operations. This is much different from other scheduling techniques. Other techniques schedule an operation *from a particular iteration* with previously scheduled operations from specific iterations. This technique schedules an operation from *all* iterations with previously scheduled operations from *all* iterations. In other words, one cannot schedule operation a from iteration 1 without scheduling operation a from all iterations.

Several different algorithms have been derived from the initial framework laid out by Rau et al. [1981; 1982].

2.1 Modulo Scheduling via Hierarchical Reduction

Several important improvements over the basic modulo scheduling technique were proposed by Lam [1988]. Her use of modulo variable expansion in which one variable is expanded into one variable per overlapped iteration has the same motivation as architectural support of the rotating register. Rau originally included the idea as adapted to polycyclic architectures as part of the Cydra 5, but the ideas were not published until later due to proprietary considerations [Rau et al. 1989; Beck et al. 1993]. The handling of predicates by taking the *or* (rather than the sum) of resource requirements (termed hierarchical reduction) of disjoint branches is a goal incorporated into state of the art algorithms. Hsu's [1986] stretch scheduling developed concurrently.

This algorithm is a variant of modulo scheduling in which strongly connected components[7] are scheduled separately [Lam 1988, 1987]. Although Lam uses a traditional list scheduling algorithm, several modifications must be made to create the flat schedule.

Lam's model allows *multiple* operations to be present in a given node of the dependence graph. Because her method breaks the problem into smaller problems that are scheduling separately, she needs a way to store the schedule for a subproblem at a node. Each strongly connected component is reduced to a single node, representing its resulting schedule; this graph is termed a *condensation*. Because the condensed graph is acyclic, a standard list-scheduling algorithm is used to finish scheduling the loop body.

Modifying the DDG Model. Each node in the DDG becomes a *schedule* of instructions for the subproblem instead of a single operation. Usage of resource j is indicated by placing a mark in the j^{th} column of the table. Rows of the table indicate time within the instruction group. Figure 20 shows an example of Lam's DDG model. Columns represent various resources and rows represent time. Resource usage vectors are shown in square brackets for each instruction group. Node B consists of operations scheduled in two time steps. In the first time step, resources 3 and 4 are used. The *resource usage vector* (ρ) to the right of node B indicates how many times each resource is used. $\rho(B) = [1, 0, 2, 1]$ indicating resource 3 is used two times whereas resources 1 and 4 are used only once.

The resource usage vector can be subscripted to indicate the resource usage at a given point in the schedule. For a given node v consisting of $|v|$ time steps, $\rho_v(j)$ indicates the resource usage vector utilized by node v in its j^{th} time step, for $0 \leq j < |v|$. In Figure 20, $\rho_C(0) = [0, 1, 1, 0]$ and $\rho_C(1) = [0, 1, 0, 0]$, together giving $\rho_C = [0, 2, 1, 0]$. The maximum number of each resource type available per time step is contained in the *limit resource vector*, R. For simplicity of presentation, a single resource of each type is assumed.

Scheduling the Connected Components. When scheduling a node v of a strongly connected component, the transitive closure of dependences along paths from every node in the component to node v must be considered. The shortest path cost matrix discussed in Section 1.6.2 is utilized.

Modulo scheduling uses the closure of the dependence constraints to form a range of locations in the flat schedule in which an operation must be placed. Because this range often depends on the placement of other nodes, the range for all nodes is updated after each node is scheduled. The initial dependence con-

[7] A *strongly connected component* of a digraph is a set of nodes such that there is a directed path from every node in the set to every other node in the set. Strongly connected components can be found with Tarjan's algorithm [1972].

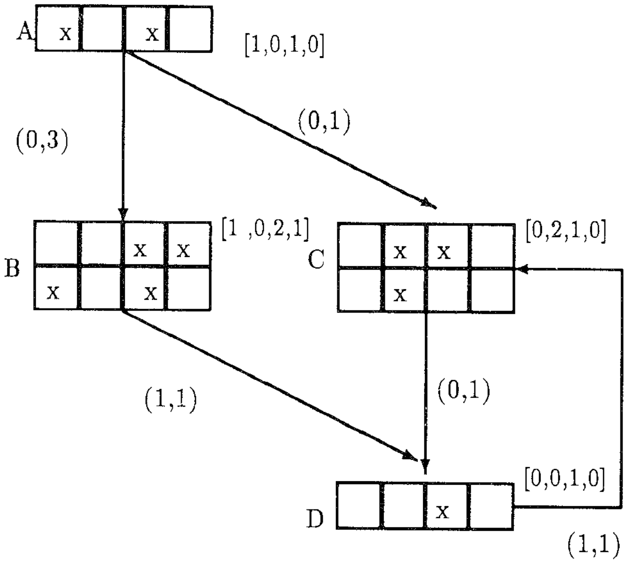


Figure 20. DDG model in which each node is an instruction group.

straint range for each node v is defined as follows:

$$\sigma_{low}(v) = \max_{(u \in N)} cost^{II}(u, v)$$

$$\sigma_{up}(v) = \infty.$$

Some care must be taken when using these initial bounds. The lower bound can be negative. Even though $\sigma_{low}(v)$ is negative, the node should be scheduled in instruction zero.

Nodes in N are scheduled in topological order of the loop-independent subgraph. A *topological order* of a graph is a sequential ordering of all nodes such that, if there is an arc from a to b, a comes before b in the order. The *loop-independent subgraph* is the graph of the strongly connected component without the loop-carried dependence arcs. A node is *data ready* if all its predecessors already have been scheduled. When there are more than two nodes data ready, the one with the lowest upper bound (σ_{up}) is chosen. When v is selected to be scheduled, it is placed in the first instruction (in the range from $\sigma_{low}(v)$ to $\sigma_{up}(v)$) that does not cause a resource conflict.[8] If the node cannot be scheduled in this range, the scheduling algorithm fails for the current initiation interval. The dependence constraint range for a node may be larger than II instructions. In this case, if the node cannot be placed in II consecutive instructions,[9] the algorithm fails for the current initiation interval. When the algorithm fails for the current initiation interval, the initiation interval is increased by one and the algorithm is retried.

Once a node v has been scheduled, the dependence constraint ranges of each remaining unscheduled node u must be updated. The new lower bound is the larger of its current value and the placement location caused from the arc from v to u (where v has just been placed, and thus $\sigma(v)$ is known). Similarly, the new

[8] Other scheduling strategies may be beneficial [Huff 1993]

[9] Because the schedule is cyclic, all cyclic locations are considered by looking at II consecutive locations in the flat schedule. If an operation conflicts with operations in II successive instructions, all *cyclic* instructions have been examined and there is no point in continuing.

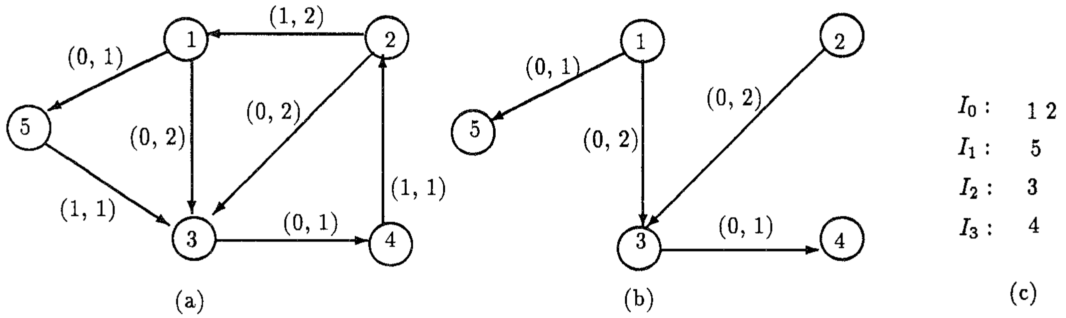


Figure 21. (a) Example of strongly connected component. (b) Loop-independent subgraph. (c) Schedule.

Table 3. Closure of Dependence Constraints for Strongly Connected Component

Source Node	*Destination Node*				
	1	2	3	4	5
1	{(2, 6)}	{(1, 4)}	{(0, 2)}	{(0, 3)}	{(0, 1)}
2	{(1, 2)}	{(1, 4)}	{(0, 2)}	{(0,3)}	{(1, 3)}
3	{(2, 4)}	{(1, 2)}	{(1, 4)}	{(0, 1)}	{(2, 5)}
4	{(2, 3)}	{(1, 1)}	{(1, 3)}	{(1, 4)}	{(2, 4)}
5	{(3, 5)}	{(2, 3)}	{(1, 1)}	{(1, 2)}	{(3, 6)}

upper bound is the smaller of its current value and the placement location caused from the arc from u to v. The following formulas are used to update the schedule range:

$$\sigma_{low}(u) = \max(\sigma_{low}(u), \sigma(v) + cost^{II}(v,u)),$$

$$\sigma_{up}(u) = \min\big(\sigma_{up}(u), \sigma(v) - cost^{II}(u,v)\big).$$

Consider the strongly connected component of Figure 21(a). For the sake of simplicity, assume that all operations in the loop can execute concurrently without resource conflicts. The loop is a single strongly connected component. The first step in scheduling the component is calculating the closure of the dependence constraints.

Table 3 shows C, the result of calculating the closure of the dependence constraints. The initial dependence constraint ranges computed from $\sigma_{low}(v) = \max_{(u \in N)} cost^{II}(u, v)$ and $\sigma_{up}(v) = \infty$ (where II is initially 4) are as follows:

n_1: $[-2, \infty]$,
n_2: $[0, \infty]$,
n_3: $[2, \infty]$,
n_4: $[3, \infty]$,
n_5: $[1, \infty]$.

The preceding example illustrates a case where the initial dependence constraint range has a negative lower bound.

The loop-independent subgraph, shown in Figure 21(b), indicates that both n_1 and n_2 are initially data ready as they have no predecessors. Both nodes have an upper bound of ∞, so n_1 is arbitrarily chosen to be scheduled first, and is scheduled in I_0. Because of the placement of n_1, the dependence constraint ranges are updated as follows:

n_2: $[0, 2]$,
n_3: $[2, 4]$,
n_4: $[3, 5]$,
n_5: $[1, 7]$.

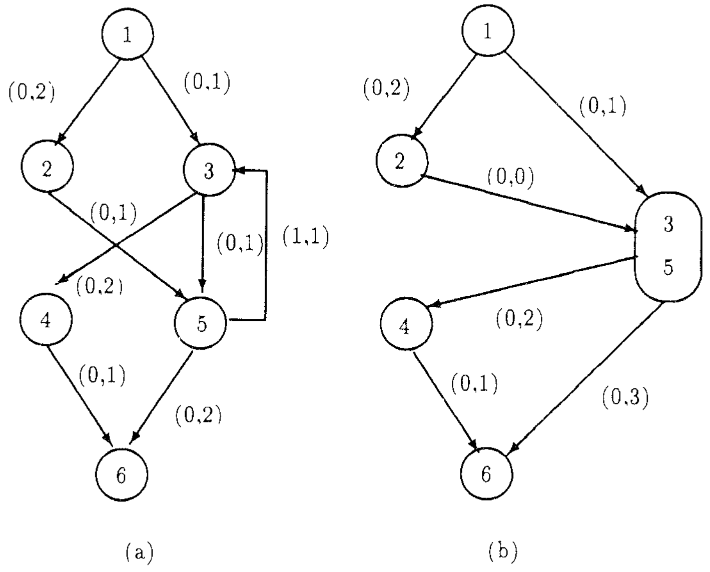


Figure 22. Example of reducing strongly connected component.

Because n_1 has been scheduled, both n_2 and n_5 are data ready. Note n_2 has a smaller upper bound than n_5, so it is scheduled first. It is also scheduled in I_0. The updated dependence constraint ranges are as follows:

n_3: [2, 2],
n_4: [3, 3],
n_5: [1, 5].

Nodes n_3 and n_5 are now data ready. Node n_3 has the lower upper bound and is scheduled in I_2. The updated dependence constraint ranges are as follows:

n_4: [3, 3],
n_5: [1, 5].

Now, nodes n_4 and n_5 are data ready. Node n_4 has the lower upper bound and is scheduled in I_3. The final dependence constraint range is

n_5: [1, 5].

Finally, n_5 is scheduled in I_1. The final schedule, shown in Figure 21(c), produces a legal execution order when used as a flat schedule. As can be seen in this example, the dependence constraint ranges for a node shrink as the scheduling process proceeds. The narrowing of the dependence constraint ranges reflects the increased constraints placed on a node as more nodes in the strongly connected component are scheduled.

Reducing the DDG. After each strongly connected component is successfully scheduled, it is condensed to a single node as follows. The schedule of the strongly connected component becomes the contents of the new node. Because resources are represented as a reservation table (as shown in Figure 20), the resource requirements of a composite node are easy to represent. The minimum delay on arcs entering or leaving the condensed node is also changed so the delay is measured with respect to the first time step of the new node. This is necessary as this model can only represent the minimum time between the beginning of one condensed node to the beginning of another. Thus an arc specifying that the m^{th} instruction of node A must precede the n^{th} instruction of node B by k must be formulated as the beginning of node A must precede the beginning of node B by $k + m - n$.

Figure 22 illustrates the process of reducing a strongly connected component. Figure 22(a) shows a DDG with a strongly connected component {3, 5}. Assume the following schedule results from scheduling the strongly connected component: $\sigma(3) = 0$, and $\sigma(5) = 1$. Figure 22(b) shows the DDG after the strongly connected component has been reduced to a single node. The node labeled with both 3 and 5 represents the node resulting from the reduction of the strongly connected component. The arcs $3 \rightarrow 5$ and $5 \rightarrow 3$ are removed because both are contained (and satisfied) in the condensed node. The arc from $2 \rightarrow 5$ is changed from $(0, 1)$ to $(0, 0)$.

Scheduling the Acyclic DDG. Once all strongly connected components are scheduled, the nodes of the condensed graph are scheduled in topological order of the DDG. The priority of a node resulting from a strongly connected component is defined as its height[10] in the DDG plus the maximum height of the DDG. Due to the weighting, this ensures that nodes resulting from a graph reduction are always scheduled before other nodes whenever possible. As these composite nodes have more resource constraints, they are more difficult to schedule and need to be scheduled early.

Once a node n is selected for scheduling, the earliest instruction in which it can be placed without violating dependence constraints is determined by its distance from previously scheduled nodes. This lower limit on placement is given by the following formula:

$$\sigma_{low}(n) = \max_{((a \rightarrow n, dif, min) \in E \mid a \in Scheduled)} (\sigma(a) + min - II * dif),$$

where *Scheduled* is the set of nodes already scheduled. The formula must take into account the iteration difference on arcs because there still may be some loop-carried dependences that are not contained within a strongly connected component. The node is placed in the first instruction between $\sigma_{low}(n)$ and $\sigma_{low}(n) + II - 1$ that does not cause a resource conflict. If the entire DDG is scheduled, the result is a schedule for a single iteration that forms a regular pipeline, when used as a flat schedule. If the node cannot be placed in any instruction in this range, the scheduling algorithm fails for the current initiation interval. Then II is incremented and the entire process is repeated until a regular schedule is achieved or the upper bound on II is exceeded.

Pipelining the Loop. The software pipelining algorithm can be summarized as follows. The strongly connected components of the DDG are found. Then, for the nodes in each strongly connected component, the closure of the dependence constraints is computed. The algorithm calculates an absolute lower bound of II for the entire graph. The upper bound is the length of the loop body when it is compacted without pipelining constraints. The lower bound is found by taking the maximum of the lower bound estimate due to resource constraints and the lower bound estimate due to dependence constraints. In other words, because every lower bound simply means we know the schedule cannot be any tighter, the smallest initiation interval possible is the largest of the various lower bounds.

The scheduling process proceeds in two steps. First, each strongly connected component is scheduled and reduced to a single node. Thus an acyclic DDG is produced. The second step schedules the acyclic DDG to produce $\mathscr{F}$. If either of these scheduling processes fail, the algorithm cannot find a regular pipeline for the current II. If this happens, the original DDG is restored, and the scheduling process is repeated after incrementing II. If the algorithm cannot create a regular pipeline with an initiation interval less than or equal to the upper bound on the initiation interval, the algorithm fails and software pipelining is ineffective for this loop.

[10] The height of a node in a DDG is the length of the longest path from the node to a sink. The height of a graph is the maximum height of any node.

Operations that are overlapped with a condition compete for resources with the *union* of the requirements on the branches rather than with the sum of the resource requirements,[11] which is a feature adopted by later modulo scheduling algorithms. An important drawback is the fact that schedules may not respond appropriately to a larger *II*. As *II* is increased, instead of causing the operations to spread further apart, operations tend to remain clustered as in the previous schedule using a smaller *II*. Inefficiencies in the code are also introduced by scheduling strongly connected components separately. The problems of hierarchical scheduling (originally proposed by Wood [1979]) are addressed in the Enhanced Modulo Scheduling algorithm [Warter et al. 1992].

2.2 Path Algebra

Path algebra is an attempt to formulate the software pipelining problem in rigorous mathematical terms [Zaky 1989]. In Section 1.6.3, path algebra was used to determine a viable *II* using the matrix *M*. This same matrix also can be used to determine a modulo schedule for software pipelining. Nodes that are on the critical cycle (having maximum *min/dif*) have a zero on the diagonal of $\Gamma(M)$ indicating the node must be *exactly* zero locations from itself.[12] Furthermore, each row in the matrix $\Gamma(M)$ indicates the relative placement of nodes with respect to each other. A row that has a zero on the diagonal is a solution to the algebraic equation regarding distances and is termed an *eigenvector*. Consider the graph of Figure 11 and the corresponding closure matrix, as shown in Figure 12(c).

In this case, the first five rows have a zero on the main diagonal as those nodes are involved in a critical cycle of the graph. The row gives the relative placement of a node with respect to the element on the diagonal. Row 1 is (0, 1, 0, 0, 1, 2, 3) which indicates that if O_1 is scheduled in the *0*th location, a legal schedule is formed if O_2 is in the 1st location, O_3 is in the 0th, and so on. The first five rows give the same *relative* placement. Row 2 is $(-1, 0, -1, -1, 0, 1, 2)$ which indicates that if O_1 is scheduled in the -1st location, O_2 must be in the 0th location, O_3 in the -1st, and so on. Notice that the elements of row 2 differ by a constant from the elements of row 1, hence the relative placement of operations indicated by each row is identical.

To understand the resulting matrix, let us reconsider alternative estimates for *II*. If *II* is one, the original matrix and the closure are shown in Figure 23. We can tell that $II = 1$ is not sufficient to guarantee a correct schedule because of the distances on the main diagonal. For example, the distance between $1 \rightarrow 1 = 3$, meaning O_1 must follow O_1 by at least three instructions. Obviously this cannot happen when *II* is 1. Figure 23(c) shows the schedule deduced from the first line. This schedule is incorrect as, for instance, the arc from five to one is not obeyed.

With $II = 3$, as shown in Figure 24, all diagonals are negative in the closure graph. For instance, O_1 must follow O_1 by at least -4 time units. In other words, O_1 can *precede* O_1 from the next iteration by 4 time units. Each row indicates a different schedule, but all are legal.

This solution is elegant, but cannot handle resource requirements. As such, it becomes a theoretical tool rather than a practical one.

2.3 Predicated Modulo Scheduling

Predicated modulo scheduling has all the advantages of other techniques discussed in this section, but represents an improvement of known defects. It is an ex-

[11] This is an improvement over early predicated execution methods. Note that there is a trade-off between a smaller *II* and a smaller code size.

[12] This is a little confusing in that it seems obvious that *every* node must be exactly zero locations from itself or *II* locations from itself in the next iteration. The point is that if dependence constraints *force* this distance, the techniques of path algebra can compute the required schedule.

	1	2	3	4	5	6	7
1	$-\infty$	1	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
2	$-\infty$	$-\infty$	0	$-\infty$	$-\infty$	1	$-\infty$
3	$-\infty$	$-\infty$	$-\infty$	1	$-\infty$	$-\infty$	$-\infty$
4	$-\infty$	$-\infty$	$-\infty$	$-\infty$	1	$-\infty$	$-\infty$
5	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
6	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	1
7	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-1	$-\infty$

(a)

	1	2	3	4	5	6	7
1	3	4	4	5	6	8	9
2	2	3	3	4	5	7	8
3	2	3	3	4	5	7	8
4	1	2	2	3	4	6	7
5	3	4	4	5	6	8	9
6	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	0	1
7	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-1	0

(b)

1	
2,3	1
4	2,3
5	4
	5
6	
7	6
	7

(c)

Figure 23. (a) Original matrix. (b) Closure. (c) Derived schedule using row 1 for $II = 1$.

cellent technique that has been implemented in commercial compilers.

Many researchers have embraced modulo scheduling for architectures with hardware support for modulo scheduling[13] and have modified the resulting code to work on architectures without hardware support [Warter et al. 1993]. The Cydra 5 work is described in Dehnert et al. [1989] and Dehnert and Towle [1993]. We use the term *Predicated Modulo Scheduling* to represent this general category of algorithms. In all but Huff [1993], the precise method for scheduling operations is not discussed, probably because of the complexity of explaining the process. One must assume the method used is similar to that employed by Lam except that the hierarchical reduction of schedules produced for strongly connected components (which generates suboptimal results) is circumvented.

Register Renaming. When iterations are overlapped, the reuse of registers becomes a concern. In the example of Figure 25(a) suppose operation 1 writes to a register (call it x) that is not used for the last time until operation 6 of the same iteration, and operation 3 writes to a register (call it y) that is not used for the last time until operation 7. In the data dependence graph, these lifetimes manifest themselves not only in the dependence chain from 1 to 6 and from 3 to 7, but also in the antidependences of 7 → 3 and 6 → 1 (shown by dotted lines in the graph). The antidependences have a $(1, 0)$ annotation indicating they are loop-carried ($dif > 0$) and that the operation which writes to the register can be executed in the *same* instruction as the last use ($min = 0$). This is possible if we assume the fetch of a value precedes the write within a machine cycle. The dependence from operation 1 to 2 has a $(0, 2)$ annotation indicating that the operation takes two cycles to complete ($min = 2$). The antidependences force the maximum min/dif to be 4. Thus the schedule shown in Figure 25(b) is of length 4. Let II_{anti} be the length of the longest cycle involved in a dependence cycle containing a loop-carried antidependence. Let II be the initiation interval for the schedule when antidependences are ignored. If we replicate the loop so that there are II_{anti}/II copies of the loop body, we can use different registers in each copy. This replicated loop is scheduled and shown in Figure 25(c). Operations that write to different registers are indicated with a

[13] See Dehnert et al. [1989], Huff [1993], Mahlke et al. [1992], Rau et al. [1992], Rau and Fisher [1993], Rau et al. [1989], Tirumalai et al. [1990], and Warter et al. [1992].

	1	2	3	4	5	6	7
1	$-\infty$	1	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
2	$-\infty$	$-\infty$	−2	$-\infty$	$-\infty$	1	$-\infty$
3	$-\infty$	$-\infty$	$-\infty$	0	$-\infty$	$-\infty$	$-\infty$
4	$-\infty$	$-\infty$	$-\infty$	$-\infty$	−1	$-\infty$	$-\infty$
5	−2	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
6	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	1
7	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	−5	$-\infty$

(a)

	1	2	3	4	5	6	7
1	−4	1	−1	−1	−2	2	3
2	−5	−4	−2	−2	−3	1	2
3	−3	−2	−4	0	−1	−1	0
4	−3	−2	−4	−4	−1	−1	0
5	−2	−1	−3	−3	−4	0	1
6	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	−4	1
7	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	−5	−4

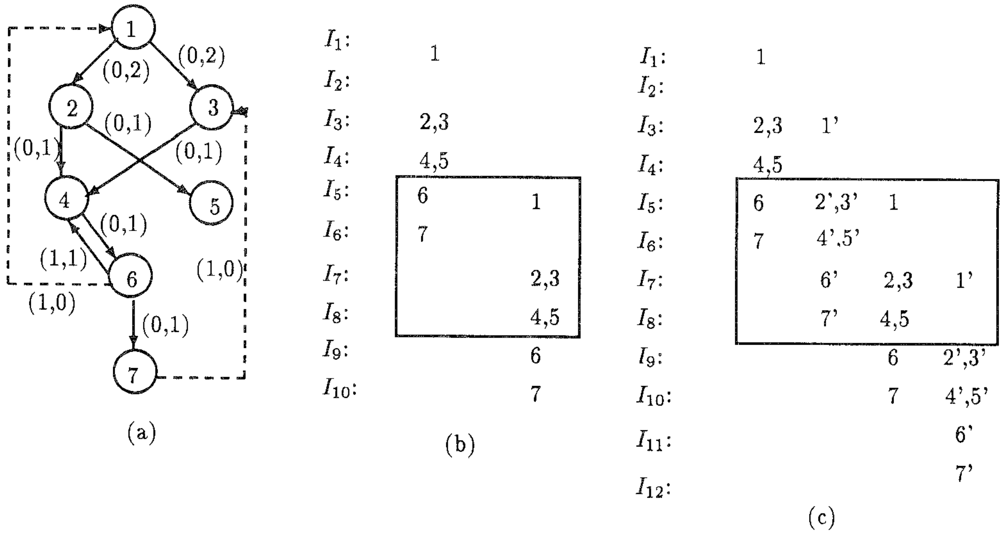
(b)

Time	Row 1		Row 2		Row 3		Row 4		Row 5	
0	1		1		3		3,4		5	
1			2		1		1		3,4	
2	5		5		2		2		1	
3	3,4	1	3,4	1	5,6	3	5,6	3,4	2	5
4				2	4,7	1	7	1	6	3,4
5	2	5		5		2		2	7	1
6	6	3,4	6	3,4		5,6		5,6		2
7	7		7			4,7		7		6
8		2								7
9		6		6						
10		7		7						

(c)

Figure 24. (a) Original matrix. (b) Closure. (c) Derived schedule using various rows for $II = 3$.

prime (e.g., 1′). In this case, as there are two copies of each operation, there are two versions of each register whose lifetime extends beyond II. Table 4 shows the same schedule with renamed versions of a register differing by a prime. Writes of an operation are shown by the register name appearing to the left of an equal sign. Reads from a register are shown by the register name appearing to the right of an equal sign. For simplicity, only registers x and y are shown. Operation 1 writes to x in the first iteration and x' in the second iteration. In the third iteration, x is used again. Similarly, operation 3 writes to y in the first and third iterations and writes to y' in the second iteration. Even though I_5 uses x and writes x, there is no problem as fetches precede stores within the cycle. Instead of two registers, four (x, x', y, y') are required and code space has increased, but the effective initiation interval is halved as there are two versions of



I_1:	1	
I_2:		
I_3:	2,3	
I_4:	4,5	
I_5:	6	1
I_6:	7	
I_7:		2,3
I_8:		4,5
I_9:		6
I_{10}:		7

(b)

I_1:	1			
I_2:				
I_3:	2,3	1'		
I_4:	4,5			
I_5:	6	2',3'	1	
I_6:	7	4',5'		
I_7:		6'	2,3	1'
I_8:		7'	4,5	
I_9:			6	2',3'
I_{10}:			7	4',5'
I_{11}:				6'
I_{12}:				7'

(c)

Figure 25. (a) DDG. (b) Schedule. (c) Schedule after renaming to eliminate loop-carried antidependence.

Table 4. Modulo Variable Expansion

Time	Iterations			
I_1	1 ($x =$)			
I_2				
I_3	2,3 ($y =$)	$1'$ ($x' =$)		
I_4	4,5			
I_5	6 ($= x$)	$2', 3'$ ($y' =$)	1 ($x =$)	
I_6	7 ($= y$)	$4', 5'$		
I_7		$6'$ ($= x'$)	2,3 ($y =$)	$1'$ ($x' =$)
I_8		$7'$ ($= y'$)	4,5	
I_9			6 ($= x$)	$2', 3'$ ($y' =$)
I_{10}			7 ($= y$)	$4', 5'$
I_{11}				$6'$ ($= x'$)
I_{12}				$7'$ ($= y'$)

each original operation in the kernel. This is termed *modulo variable expansion* [Lam 1988]. Note that modulo variable expansion is a code expansion that occurs after scheduling (and hence does not increase complexity), rather than an expansion that occurs before scheduling (as does replication).

One drawback of this scheme is that only loops that execute a multiple of $(span - u) + u * i$ times can be accommodated, where u is the number of copies of the loop, $span - u$ is the number of copies of the loop that are present in the prelude and postlude, and $i \geq 0$. In this case, the prelude and postlude execute two iterations, and every execution of the new loop body completes two original iterations so only loops that execute $2 + 2i$ iterations can be handled. In some situations in which the new loop body contains multiple copies of the original iteration, a branch out of the loop body avoids this restriction. In this case, a

Table 5. Rotating Register File

	II = 2									
	Time									
Register	1	2	3	4	5	6	7	8	9	10
3			x^2	x^2	x^2	x^2	$*y^3$	y^3	y^3	y^3
2					x^3	x^3	x^3	x^3	$*y^4$	y^4
1			y^1	y^1	y^1	y^1	x^4	x^4	x^4	x^4
0	x^1	x^1	x^1	x^1	$*y^2$	y^2	y^2	y^2	x^1	x^1

jump out of the middle of the loop would require a special postlude as the registers used in the current postlude would not be appropriate.

One solution is to execute m iterations before entering the pipelined loop. m is chosen so that the remaining number of iterations to be completed is of the form $(span - u) + u * i$. This is termed *preconditioning* the loop [Rau et al. 1992]. According to Rau et al., this is acceptable for architectures with little instruction level parallelism, but is inadequate for more powerful processors.

Hardware support for modulo scheduling simplifies register renaming. With the advent of rotating register files ([Rau et al. 1992, 1989]), loop-carried antidependences can be ignored without code expansion. Variables that are not redefined in the loop or whose lifetime is less than *II* can be assigned static general purpose registers. Variables involved in loop-carried dependence cycles can take advantage of *rotating register files*. A rotating register file is a file whose file pointer rotates. With a rotating register file, register specifier n does not always refer to the same physical register, but rotates over the set of registers. This is accomplished by treating the register specifier as an offset of the *Iteration Control Pointer* (ICP) that points to the beginning of the registers for the current iteration. Every register reference is computed as the sum of the register specifier and the ICP (modulo the register file size). The ICP is decremented (modulo register file size) at the end of each iteration execution. In this way, each iteration accesses different registers even though the code remains identical for each iteration. This provides hardware managed renaming.

Using a rotating register file for our example, the schedule of Figure 18(b) (rather than Figure 25(c)) is achieved. Operation 1 always writes the same register specifier (0 in this case), but because the registers rotate there is no problem. Similarly, operation 2 always writes to register specifier 1.[14] In Table 5, the rows of the table represent four registers labeled 0 through 3. Time progresses horizontally. The superscripts represent the original iteration defining the variable. An asterisk indicates that the register contains the value of x at the beginning of the cycle and the value of y at the end of the cycle. Even though the number of registers required is the same using rotating register files or modulo variable expansion, register assignment constantly changes (as opposed to being fixed throughout the loop). Register 0, for example, stores the x from iteration 1 for the first five cycles, and the y from iteration 2 until the end of the eighth cycle. Notice that at the beginning of each new *II*, the location of x and y for that cycle is decremented (modulo 4). Because the ICP was initially 0, it is decremented to 3 $(0 - 1 \bmod 4)$ before the second iteration and thus x^2 (with

[14] In general, register specifiers are not adjacent but are spaced to reflect the lifetime of the registers

register specifier 0) is assigned to physical register 3.

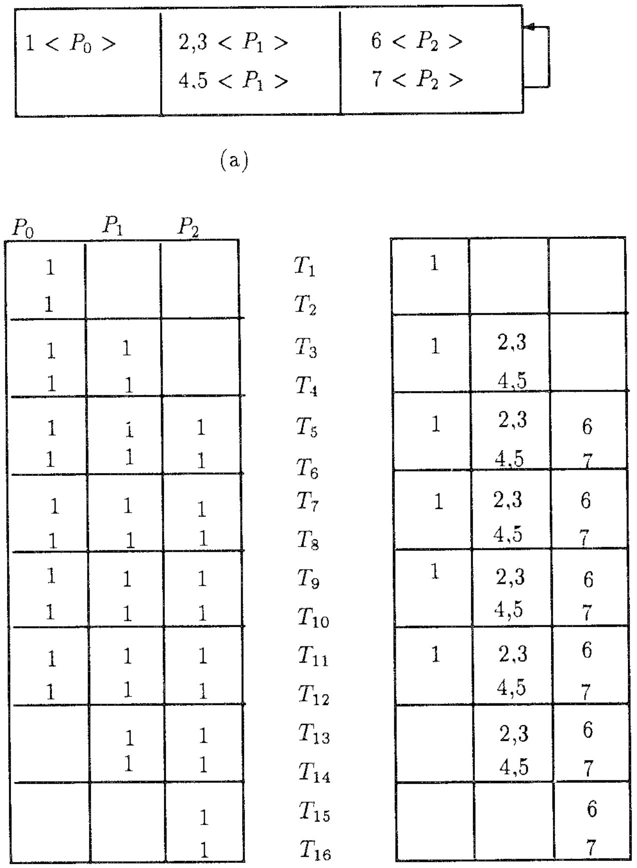
Predicated Execution. When code contains conditionally executed code, modulo scheduling becomes more complicated. Consider the example of Figure 18. Suppose that operation 2 computes a predicate (Boolean value) that determines whether operations 4 and 6 or operations 5 and 7 should be executed. If the predicate is true, operations 4 and 6 are executed. If the predicate is false, operations 5 and 7 are executed. Clearly the schedule of Figure 18(b) is illegal as 4 and 5 are never both executed. We would need two versions of the code *for each iteration* of the replicated schedule. Because code from three iterations is overlapped, there could be eight combinations resulting in complicated code expansion. Let the notation (true, false, true) correspond to the values of the predicate in successive iterations being true, then false, then true. Clearly there are eight combinations of three Boolean values. One solution to this problem is termed *Hierarchical Reduction* [Lam 1987]. Instead of scheduling each branch of a conditional separately, the code for both branches is scheduled with the understanding that only one of the branches is actually executed at run time. In other words, the schedule is created so that it is legal regardless of which branch is taken; the needs of both branches are considered. The resource conflicts must be adjusted so that at any point in time, the union[15] of the resources required by different branches is available rather than requiring the sum of the resources to be available. Even though only one copy of the pipeline is needed during scheduling, physically the eight copies still exist.

A hardware implementation of this idea involves the use of *predicated execution*. Predicated execution makes it possible to execute an operation conditionally. Instead of jumping around an operation that should not be executed, the hardware can just ignore the effects of the operation. For example, the floating point multiply specified by $r1 = fmpy(r2, r3)$ $\langle p1 \rangle$ is executed only if predicate $p1$ is true. If $p1$ is false, either the operation is ignored (treated as a no-op) or is executed but the result register is not changed. The former implementation saves effort, but the latter implementation allows the operation to be executed in the same instruction as the predicate is computed. As the results of predicate evaluation are known before the target register is written, such overlap is possible. Such hardware support may eliminate a physical jump. It also makes it possible to modulo schedule loops containing conditionals without code expansion as well as reduce the code expansion caused by a distinct prelude and postlude.

Like registers, predicates are also stored in a rotating file. Predicates can either share the regular rotating register file or have a dedicated predicate register file [Rau et al. 1992]. The process of converting code into predicate code is termed *if-conversion* and is an integral part of Enhanced Modulo Scheduling (EMS) [Warter et al. 1992]. Conditional branches are removed and control dependences become data dependences as conditionally executed operations are data dependent on the operation that generates the predicate on which they depend. This technique has more flexibility than Hierarchical Reduction in that Hierarchical Reduction completely schedules the conditional code before it attempts to schedule other operations. An arbitrary decision made in scheduling the branch can have a negative impact on the placement of other operations that are not even considered during this prescheduling phase.

In predicated execution schemes, all operations in an instruction are fetched and only those with true predicates complete execution. However, in EMS and Hierarchical Reduction, one is limited by the number of operations that physically

[15] The term union is used to indicate that two operations that cannot both execute (due to opposite values of their predicates) do *not* compete for resources.



$1 < P_0 >$	$2{,}3 < P_1 >$ $4.5 < P_1 >$	$6 < P_2 >$ $7 < P_2 >$

(a)

P_0	P_1	P_2	
1			T_1
1			T_2
1	1		T_3
1	1		T_4
1	1	1	T_5
1	1	1	T_6
1	1	1	T_7
1	1	1	T_8
1	1	1	T_9
1	1	1	T_{10}
1	1	1	T_{11}
1	1	1	T_{12}
	1	1	T_{13}
	1	1	T_{14}
		1	T_{15}
		1	T_{16}

(b)

T_1	1		
T_2			
T_3	1	2,3	
T_4		4,5	
T_5	1	2,3	6
T_6		4,5	7
T_7	1	2,3	6
T_8		4,5	7
T_9	1	2,3	6
T_{10}		4,5	7
T_{11}	1	2,3	6
T_{12}		4,5	7
T_{13}		2,3	6
T_{14}		4,5	7
T_{15}			6
T_{16}			7

(c)

Figure 26. (a) Kernel-only code. (b) Predicate values over time (c) Operations enabled by predicates

execute for a given predicate value rather than the number of operations for all predicate values. In both cases, operations that execute under disjoint predicate values can be scheduled at the same time even if they require the same nonsharable resource. Proponents of this method are so enthusiastic they even recommend using this same technique for processors that do not support predicated execution. This technique, called *reverse if-conversion*, simplifies the process of global scheduling [Warter et al. 1993]. EMS has an advantage of Hierarchical Reduction in that no prescheduling of paths is done. This increased flexibility produces superior code.

A side effect of predicated execution is that one avoids having special instructions for prelude and postlude. This is termed *kernel-only code* [Rau et al. 1992]. Figure 26(a) shows the code that requires no distinct prelude or postlude. Rather than having distinct prelude and postlude that executes a subset of the kernel operations, the execution of various stages in the pipe are controlled via predicates. The notation $1\langle P_0 \rangle$ indicates operation 1 executes if P_0 is true. Each stage of the pipeline is associated with a different predicate.

Predicates are set by the operation that jumps to the top of a loop. The predicate file rotates by decrementing the ICP (just like the regular rotating register file). The predicate register file is shifted every *II* time steps. The code to set the predicates always sets register specifier 0, but because the physical register changes, all predicates are eventually affected. This logical predicate 0 (P_0) is set to true in the prelude and is set to false in the postlude.

If the code for our example is executed six times, the predicates take on values indicated by Figure 26(b). P_0 is the logical predicate pointed to by ICP rather than the physical register. Notice that in the time step 3 the ICP has been decremented so the old value of P_0 becomes P_1 and the new logical P_0 is set. Thus both are now true. Figure 26(c) shows the operations that are executed at each point in time.

2.4 Enhanced Modulo Scheduling

Although all modulo scheduling techniques are basically the same, they differ in how they handle predicates. Hierarchical Reduction schedules operations on each branch of a conditional construct before combining using the union of the requirements. This prescheduling and unioning of requirements creates complicated pseudo-operations that are difficult to schedule efficiently with other operations. Enhanced Modulo Scheduling uses if-conversion to convert all operations into straight line, predicated code. In this form, scheduling is done noting that disjoint operations do not conflict. There is no need to preschedule the various parts of the conditional construct. After modulo scheduling, modulo variable expansion is used to rename registers' lifetimes from distinct iterations.

Predicated execution has the disadvantage that all operations from taken and untaken branches are executed (even if the results are just thrown away). Thus the resource requirement is the sum of the requirements of each branch. Enhanced Modulo Scheduling recreates the branching structure, termed *reverse if-conversion*, by inserting conditional branch instructions to eliminate predicated execution. In other words, predicated execution is used to allow operations to be scheduled independently of the branching structure and then the branching structure is reinserted. This method has some real benefits in terms of simplicity, but is also hampered by the fact that such a technique is prone to code explosion. If a predicated operation (predicated by p) happens to be scheduled early, all imperative operations that are between the first predicated operation scheduled and the last operation predicated on p must be cloned to appear on each branch. If code that is predicated by p is overlapped n times, there can be code expansion of order 2^n. Clearly this is unacceptable. Various techniques are employed to limit code explosion, the most important being the restriction of which blocks are scheduled together. The term *hyperblock* is used to denote which set of blocks are scheduled together [Warter et al. 1992, 1993].

The initial simplicity of scheduling without regard to predicates results in the complexity of introducing conditionals back into the code. Because the insertion of branches is done after code scheduling, various code inefficiencies can be introduced as is common whenever phases are segmented. According to Warter et al. [1992], Enhanced Modulo Scheduling performs 18% better than Hierarchical Reduction with up to 105% increase in code size. Although some of the increase in code size undoubtedly results from the fact that tighter code overlaps more conditional constructs, some of the increase results from unnecessary elongation of the predicated region.

3. KERNEL RECOGNITION

Although modulo scheduling algorithms create a kernel by scheduling one iteration such that it is legal when overlapped by *II* cycles, other techniques schedule various iterations and must recognize when a kernel has been formed. Some authors term this type of software pipelining algorithm *unrolling algorithms* [Rau and Fisher 1993], but the term *kernel recognition* is more accurate as there may be no physical unrolling present. Proponents of modulo scheduling point to the need to search for a kernel as a flaw, whereas proponents of kernel recognition counter that searching for a pattern can be done with hashing[16] and is much more efficient than repeating the scheduling for various goal initiation intervals. Kernel recognition proponents also argue that the ability to achieve fractional rates effortlessly makes their algorithms superior. Modulo scheduling algorithm enthusiasts claim that the *II* rarely has to be incremented over its original minimum initiation interval [Rau 1994]. Obviously, modulo scheduling can achieve fractional rates by replicating the loop body before scheduling. However, this increases complexity and may result in full iterations in prelude and postlude (that would be removed).

A first attempt at kernel recognition techniques is to mimic what one might attempt if one were scheduling by hand. The basic idea is to look at several iterations at a time and try to combine operations that are not dependent on each other. The steps are as follows:

(1) Unroll the loop and note dependences.

(2) Schedule the various operations as early as data dependences allow.

(3) Look for a block of consecutive instructions that are identical to the blocks after it. This block represents the new loop body. Rewrite the unrolled iterations as a new loop containing the repetitive block as the loop body.

The obvious question is "What do you do if no block of repeating instructions surfaces?" The URPR (UnRolling, Pipelining, and Rerolling) software pipelining algorithm, designed by Su et al. [1986] solves this problem using ad hoc techniques to force a kernel by moving duplicate operations before or after a section of code containing all operations. This moving of operations after scheduling introduces underutilized instructions and produces inferior results. Su et al. [1987] developed an extension to the basic URPR algorithm in which loops containing multiple basic blocks, abnormal entries, and conditional exits are handled. Unfortunately, the problems of URPR are carried over into the more complicated version. Although it is interesting from an historical perspective, URPR is outperformed (in terms of target code execution time) by other techniques. The next sections explain several independent kernel recognition type software pipelining algorithms.

3.1 Perfect Pipelining

Perfect Pipelining combines code motion with scheduling. It achieves fractional rates and handles general (*dif*, *min*) pairs. Techniques to assist the formation of a pattern are somewhat ad hoc.

Historically, the effectiveness of local scheduling has been limited due to the small size of basic blocks. A new architectural model in which multiple tests can be performed within a single instruction greatly enhances the degree of parallelism achieved. Aiken and Nicolau [1988b, 1988c, 1990] introduce the Perfect Pipelining algorithm[17] for use with this more general machine model [Aiken 1988; Nicolau and Potasman 1990; Breternitz 1991]. This method is important in that it reframes the problem by changing the parameters. It answers the question, "How would software pipelining be

[16] For the general resource model, hashing must be done on an encoding of the entire state of the scheduler at each point in time

[17] Aiken does not consider Perfect Pipelining to be an *algorithm* but rather a framework upon which algorithms can be built. Thus, for every reference to "the Perfect Pipelining Algorithm," the reader should substitute "one of many possible algorithms using the Perfect Pipelining framework " The algorithm referenced is one that Aiken considers in Aiken [1988]

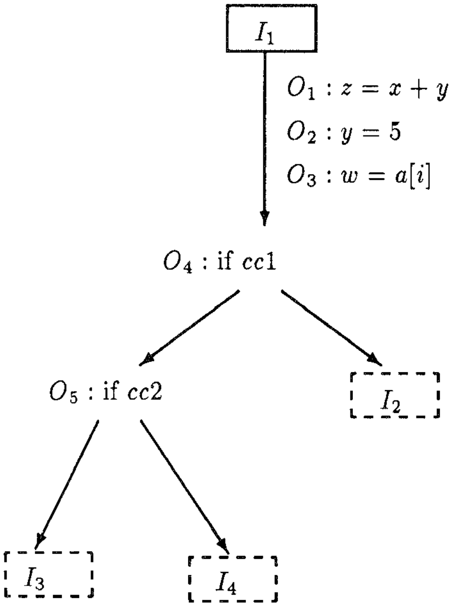


Figure 27. Perfect Pipelining's instruction model.

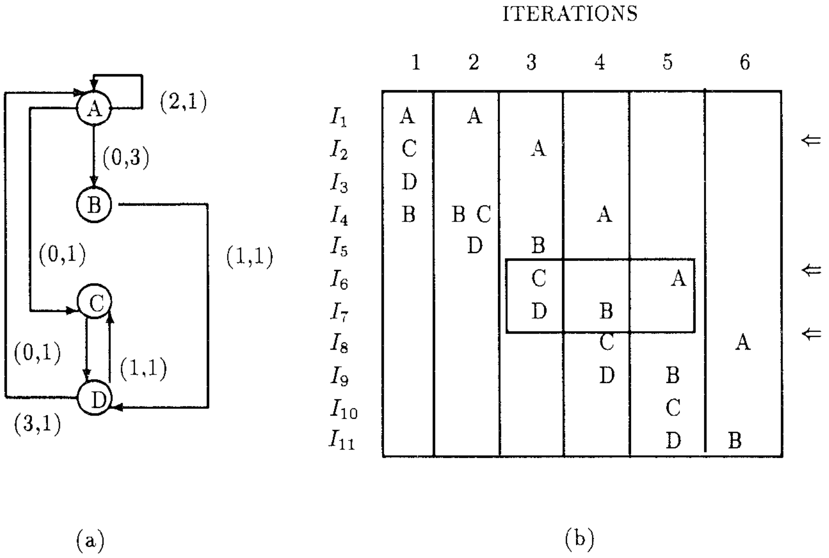
affected if the architectures were modified to support it better?"

Perfect Pipelining is somewhat similar to the method of Su et al. in that the loop is prescheduled, unrolled, and overlapped, but it is more sophisticated in that operations may move independently after the prescheduling and loops that span multiple blocks are easily accommodated. Perfect Pipelining (like Enhanced Modulo Scheduling) is more powerful than previous techniques in that it can take advantage of an architecture with multiway branching. A *multiway branching* architecture allows an instruction to have several branch target locations based on multiple Boolean conditions. Figure 27 illustrates a basic instruction executable in one time unit in this model. In one cycle, O_1, O_2, and O_3 are executed and control is transferred to one of I_2, I_3, or I_4 depending on the values of $cc1$ and $cc2$ (which are predicates set before this instruction). Such instructions are called *tree instructions* [Ebcioğlu and Nakatani 1990]. All the assignment operations are executed and a destination selected simultaneously. To take advantage of this type of architecture, a type of global code motion, called *migration*, is implemented [Aiken and Nicolau 1988a]. Migration is an improvement over early trace scheduling [Fisher 1981] in that copies can be merged, and code motion is tied to code correcting compensation for that motion directly, so the cost-benefit can be considered. Later versions of trace scheduling have adopted these improvements [Freudenberger et al. 1994].

The types of code motion are enumerated in Fisher [1981]. The addition of the operation to join multiple copies of an operation as they move past a branch point, termed *unification*, is important in that it reduces code explosion. The importance of unification is that the multiple copies of an operation generated by moving past branch points can often be recombined.

Aiken and Nicolau perform global code motion within the loop before software pipelining to simplify the initial schedule. Once global code motion has been performed, the loop is unrolled an unspecified number of times, and the result is scheduled assuming infinite resources. In a loop, performing code motion before unrolling significantly speeds up pipelining. The pipelining algorithm need not repeatedly perform similar code motions within each copy.

The assumption of infinite resources in the initial scheduling step is made because the algorithm requires unhampered motion. If the constraint of finite resources is enforced and I_1, I_2, and I_3 are sequentially ordered, motion of an operation between instructions I_3 and I_1 would be limited by the fact that the operation may not be able to temporarily reside in instruction I_2 because of resource conflicts with the operations that are placed there first. Thus instead of allowing free code motion, the algorithm would suffer from the race of which operation got to node I_2 first. The results would be somewhat history sensitive depending on the order operations moved rather than the best possible schedule.



	1	2	3	4	5	6	
I_1	A	A					
I_2	C		A				⇐
I_3	D						
I_4	B	B C		A			
I_5		D	B				
I_6			C		A		⇐
I_7			D	B			
I_8				C		A	⇐
I_9				D	B		
I_{10}					C		
I_{11}					D	B	

Figure 28. Similar nodes with different numbers of intervenmg operations.

Perfect Pipelining allows the pattern to form naturally. As each successive instruction is scheduled, one must determine whether the schedule has begun to repeat itself. Let the *state* of the schedule at a specific instruction represent the set of information that controls which operations may be scheduled in succeeding instructions. For a general resource model, state must include resources committed by the previous scheduling of operations with persistent resource requirements. For operations with nonunit latencies, state must include the concept of elapsed time between dependent operations. One must determine if the new instruction that has been generated is really necessary, or if a previous state in the schedule is the same as the current state. If two nodes can be reduced to the same state, they are said to be *functionally equivalent*. An instruction that is functionally equivalent to an earlier node can be replaced with a branch to the first instruction, thus creating a loop. When are two nodes functionally equivalent? Clearly the two nodes must look alike, but as Figure 28 demonstrates, that is not sufficient. Although I_2 and I_6 are identical, the state is different as succeeding instructions are not the same. If we assume I_2, I_3, I_4, and I_5 form a loop, operation B appears in this proposed loop three times and all other operations appear twice. Obviously, these instructions cannot form a loop as the postlude would not be static but would have to be different depending on the number of iterations actually executed. The instructions I_6 and I_7 do form a loop, which is evident from the fact that they keep repeating during the rest of the schedule.

Figure 29(b) illustrates a case in which the code between equivalent nodes represents multiple iterations of the original loop instead of exactly one iteration as seen in other algorithms. Although code space is increased, the advantage is that the code can be scheduled more tightly. For example, in Figure 29(b) the advantage of having three copies of the loop in $\mathscr{K}$ is that instead of performing one iteration in two instructions, one can achieve three iterations in five instructions. In this case, the multiple iterations represent an improvement. The ideal rate is $\frac{5}{3}$, but if we require a single iteration in the loop, the execution time is $\lceil\frac{5}{3}\rceil = 2$. We cannot achieve the ideal effective execution time with modulo scheduling unless

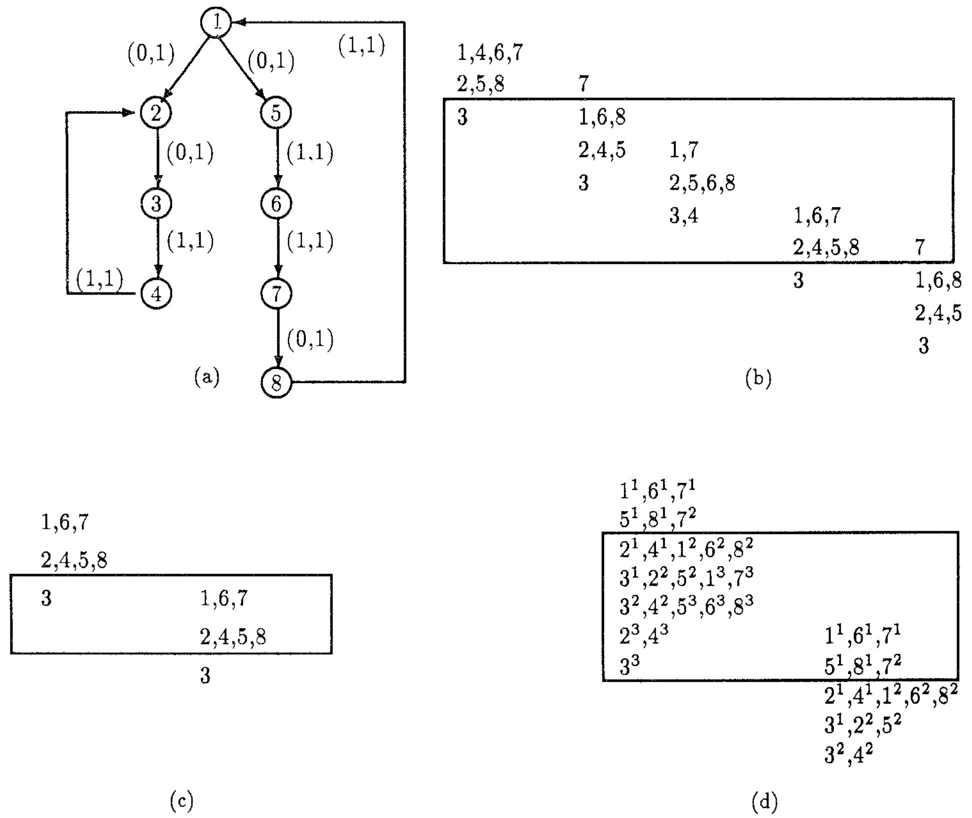


Figure 29. (a) DDG. (b) Result of Perfect Pipelining. (c) Results of modulo scheduling. (d) Result of replicating three times before modulo scheduling.

we replicate first (see Figure 29(d)). In this figure, each operation has been expanded into three operations (indicated by a superscript) due to the replication.

In other cases, multiple iterations are present in the new loop body, but no benefit is achieved. Figure 30 (a modified version of Figure 9) shows a case in which this happens. The optimal result, in terms of execution time, that is found with modulo scheduling, is shown for comparison.

Functionally Equivalent Nodes. Let $X = \{x_1^{i_1}, \dots, x_j^{i_j}, \dots, x_n^{i_n}\}$ be the set of operations x_j from iteration i_j that are scheduled at a given instruction [Aiken and Nicolau 1990]. In Figure 28(b), I_4 is referred to as the set $\{B^1, B^2, C^2, A^4\}$ and the next instruction is then set $\{D^2, B^3\}$. Two instructions are *similar* if they contain the same operations and the superscripts for the same operation differ by at most a constant. If $X = \{\dots, x_j^{i_j}, \dots\}$, let $X^c = \{\dots, x_j^{i_j+c}, \dots\}$. Although similarity is necessary for functional equivalence, it is not sufficient. Functionally equivalent nodes are two nodes that can be used interchangeably in the schedule graph. Two instructions that are similar are not always functionally equivalent. We can find two similar instructions X and Y by hashing,[18] but one has to compare the state (including resources committed by persistent irregular operations) at X and

[18] Hashing is a search technique in which a function of the key is used to determine the storage location. Two instructions that have the same contents hash to the same location, and hence are easily identified.

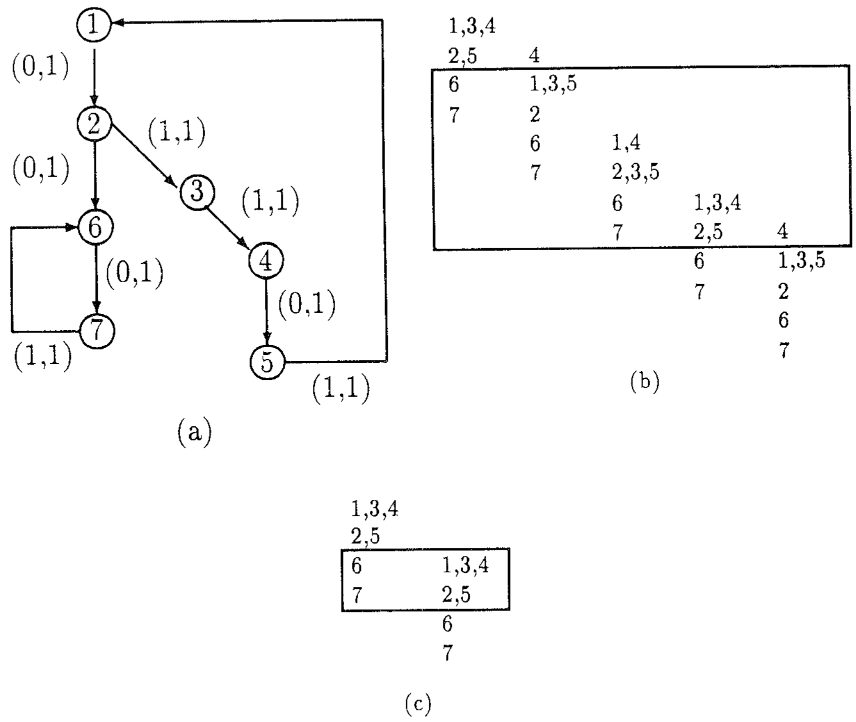


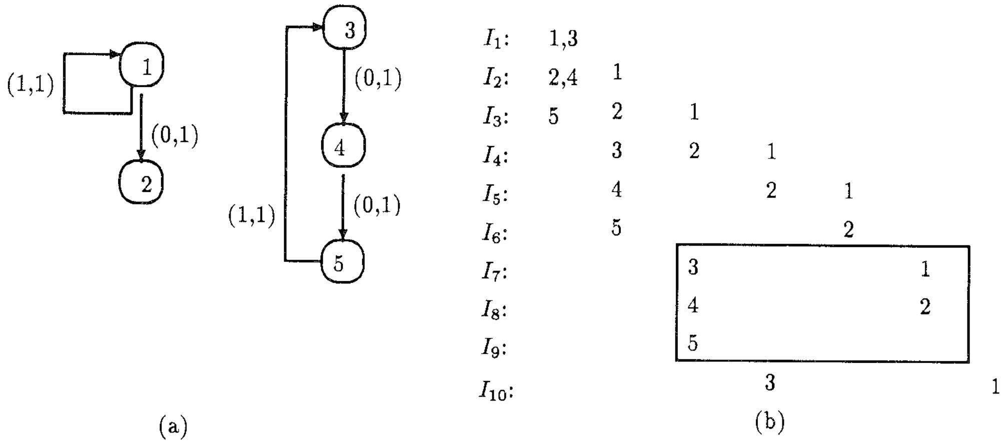
Figure 30. (a) DDG. (b) Result of Perfect Pipelining. (c) Result of modulo scheduling algorithm.

Y to ensure functional equivalence. Early versions of this algorithm were limited by the fact that true functional equivalence was not guaranteed.

For example in Figure 28, although $I_2 = \{C^1, A^3\}$ and $I_6 = \{C^3, A^5\}$ are similar, the next instructions scheduled are different; no repeating pattern has formed. In the DDG of Figure 31, we see a case where the lack of similarity indicates the nodes are not functionally equivalent. The cycle involving nodes 1 and 2 is initiated every instruction, but the cycle involving 3, 4, and 5 repeats every three instructions. Although nodes containing the same operations are found (I_1 and I_7), they are not similar in that each pair of operations in the two instructions *do not* differ by a constant subscript. Thus the procedure to find two functionally equivalent nodes must include full state information.

If we have two dependence cycles capable of different execution rates, as seen in Figure 31, we must slow the faster one down in order to get a pattern. This is accomplished by forcing the set of operations considered for scheduling to contain only operations from a fixed range of iterations. Let *Span* be the limit that indicates that until all operations from iteration i have been scheduled, operations from iteration i + *Span* cannot be scheduled. Stated another way, if operation a has been scheduled s_a times and operation b has been scheduled s_b times, then $|s_b - s_a| < Span$. The value of *Span* is experimentally determined. If *Span* is too small, a schedule does not form. If *Span* is too large, the prelude and postlude are overly complicated. The schedule of Figure 31(b) results when *Span* = 4. Notice that 1^7 is delayed until 3^3, 4^3, and 5^3 have been scheduled.

When Perfect Pipelining is applied to a loop body consisting of a single basic block with unlimited resources, a time optimal pipeline is formed. The other methods



I_1:	1,3						
I_2:	2,4	1					
I_3:	5	2	1				
I_4:		3	2	1			
I_5:		4		2	1		
I_6:		5			2		
I_7:			3			1	
I_8:			4			2	
I_9:			5				
I_{10}:				3			1

Figure 31. An acceptable pattern may not occur without intervention. (a) DDG. (b) Pipelining with *Span* = 4.

may not always form a time optimal pipeline because they force $\mathscr{K}$ to contain a single copy of each operation that may delay the start of the next iteration. On the other hand, when using the greedy scheduling algorithm, Perfect Pipelining waits for a pattern to naturally develop. Therefore, when L consists of a single basic block and resources are unlimited, it always produces a time optimal pipeline.

An Example. To see the power of this technique, one must consider branches within the loop. Consider the piece of code shown in Figure 32 (taken from Aiken [1988]) that builds a list. Each call to *append* is dependent on the previous call to *append*, and hence is serialized.

If the first seven iterations of the loop are scheduled, assuming three tests can be performed in one instruction, the execution graph of Figure 33 results in which nodes represent parallel instructions and arcs represent the flow of control. The superscript on each operation represents the iteration number of the operation. Notice there are several branches leaving each instruction. The arc labels correspond to the values of the tests. The label "fft" indicates the first two tests are false and the last test is true. An asterisk indicates a "don't care" condition. To reduce the number of different instruction formats, whenever an append is performed, the tests for the three successive iterations are executed, even if some have already been performed. Notice that the ability to perform multiple tests in a single instruction greatly reduces the execution time of the loop in cases where the append operation does not occur for every value of i. After equivalent nodes have been identified, the graph of Figure 34 shows the final result of Perfect Pipelined scheduling.

```
For i = 1 to n
    T: If B[i] = key
    A: list = append(list,i)
```

Figure 32. Sample code.

3.2 Petri Net Model

The Petri net algorithm uses the rich graph-theoretic foundation of Petri nets to solve the problem of kernel recognition. It achieves fractional rates, works for general (*dif*, *min*) pairs, and is extendible. It has the power of Perfect Pipelining, but has replaced ad hoc techniques with mathematically sound approaches.

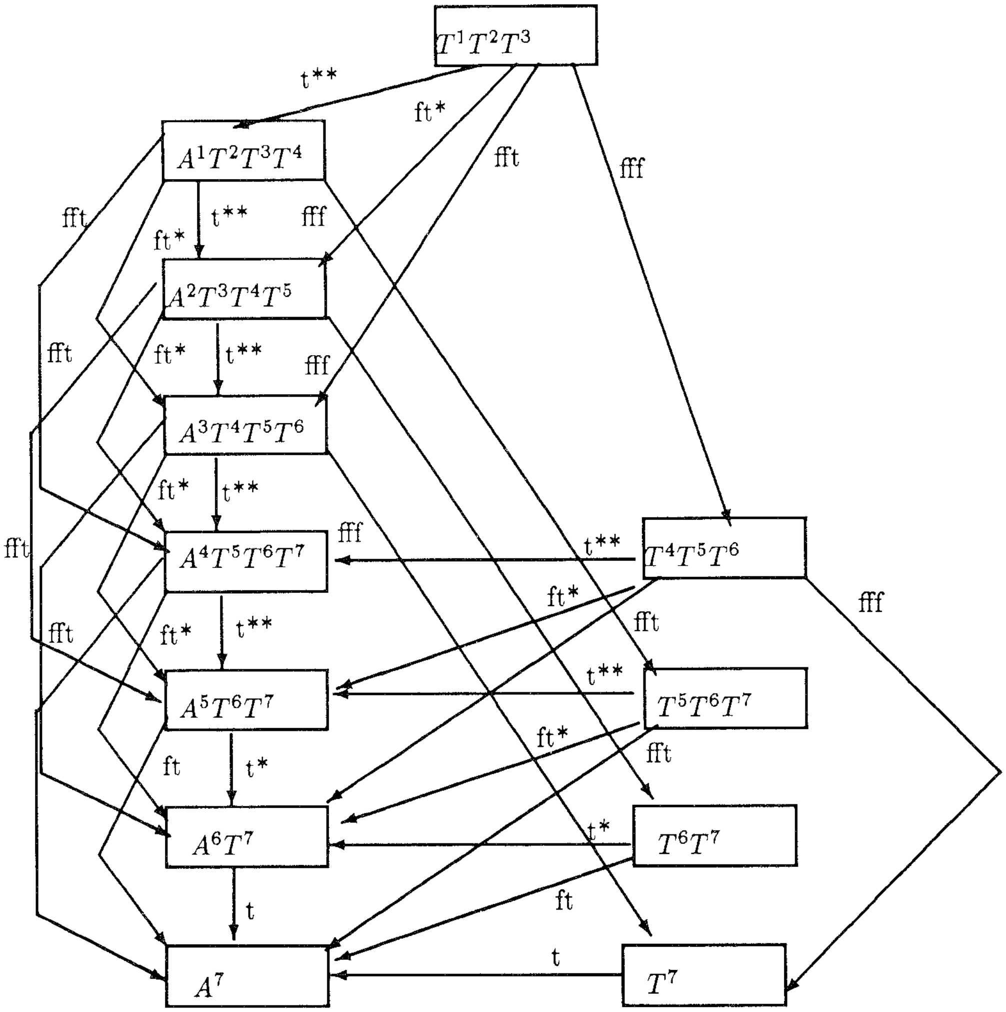


Figure 33. Loop after unrolling seven times.

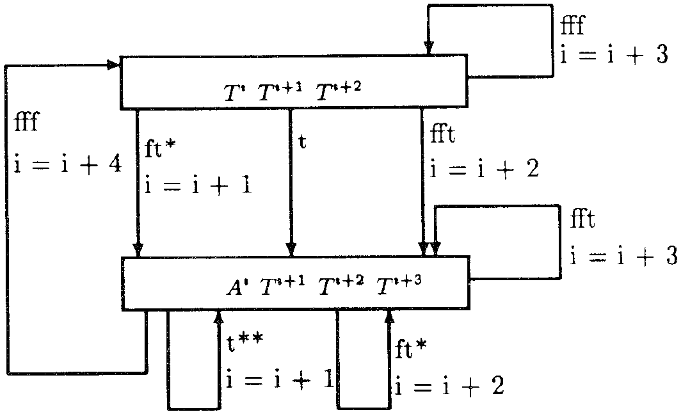


Figure 34. The same loop (Fig. 33) after pipelining.

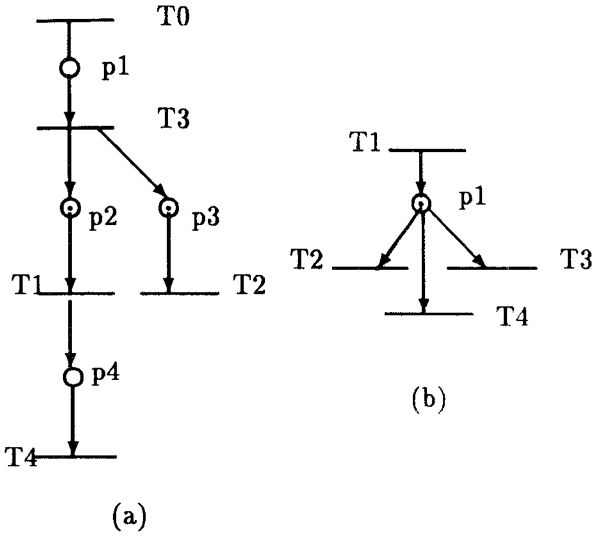


Figure 35. Petri net. (a) Concurrency. (b) Conflict.

The Petri net model of Allan, Rajagopalan, and Lee [1993] provides a valuable solution to the problems associated with the formation of a pattern, both in terms of forcing a pattern to occur and recognizing a pattern has formed [Rajagopalan and Allan 1994]. Being able to recognize when a pattern has formed and aiding the efficient formation of such a pattern is essential to kernel recognition type software pipelining. In other techniques, kernel development needs to be assisted by manipulating the final schedule and/or look-alike instructions may masquerade as loop entry points when a repeating pattern has *not* been achieved [Jones 1991]. Both problems are elegantly eliminated using Petri nets. This algorithm is an improvement over the Gau et al. [1991a, 1991b] algorithm which suffers from the following limitations:

(1) *Dif* values greater than 1 are not handled in the Gao algorithm except by replicating the code so all *dif*s are 0 or 1.

(2) Initiation intervals of less than two cannot be achieved without replication as the acknowledgment arcs create cycles and thus force an initiation interval of two.

(3) Gao's method is complicated by the addition of superfluous arcs and frequently achieves nonoptimal initiation intervals due to the fact that cycles having $min_\theta/dif_\theta > II$ are inadvertently created.

A Petri net $G(P, T, A, M)$ is a bipartite graph having two types of nodes, places P and transitions T, and arcs A between transitions and places. Figure 35(a) shows a Petri net. The transitions are represented by horizontal bars and places are represented by circles. An initial mapping M associates with each place p, $M(p)$ number of tokens such that $M(p) \geq 0$. A place p is said to be marked if $M(p) > 0$. Associated with each transition t is a set of input places $S_i(t)$ and a set of output places $S_o(t)$. The set $S_i(t)$ consists of all places p such that there is an arc from p to t in the Petri net. Similarly $S_o(t)$ consists of all places p such that there is an arc from t to p in the Petri net.

The marking at any instant defines the state of the Petri net. The Petri net changes state by *firing* transitions. A transition t is ready to fire if for all p belonging to $S_i(t)$, $M(p) \geq w_p$ where w_p

is the weight of the arc between p and t. The reader may see some similarity between transitions and runners in a relay race. One runner cannot run (fire) until he has been given the baton (token). However, in this case, a runner can pass a baton to several teammates simultaneously and one runner may have to receive a baton from each of several teammates before running.

When a transition fires, the number of tokens in each input place is decremented by the weight of the input arc while the number of tokens in each output place is incremented by the weight of the arc from the transition to that place. All transitions fire according to the earliest firing rule; that is, they fire as soon as all their input places have sufficient tokens. In Figure 35(a), there are no arcs between transitions T_1 and T_2. These transitions are independent of each other and can be fired concurrently. However, T_4 cannot fire until T_1 has fired and placed a token in place p_4. Therefore, T_4 is dependent on T_1. In Figure 35(b), place p_1 contains only one token which can be used to fire *one* of transitions T_2, T_3, or T_4. This represents a conflict that can be resolved using a suitable algorithm.

The Petri net models the cyclic dependences of a loop. A data dependence graph shows the must-follow relationship between operations (nodes). However, it cannot show which operations are ready to be scheduled at a given point in time. A Petri net is like a DDG with the current scheduling status embedded. Each operation is represented by a transition. Places show the current scheduling status. Each pair of arcs between two transitions represents a dependence between the two transitions. When the place (between the transitions) contains a token, it is a signal that the first operation has executed, but the second has not.

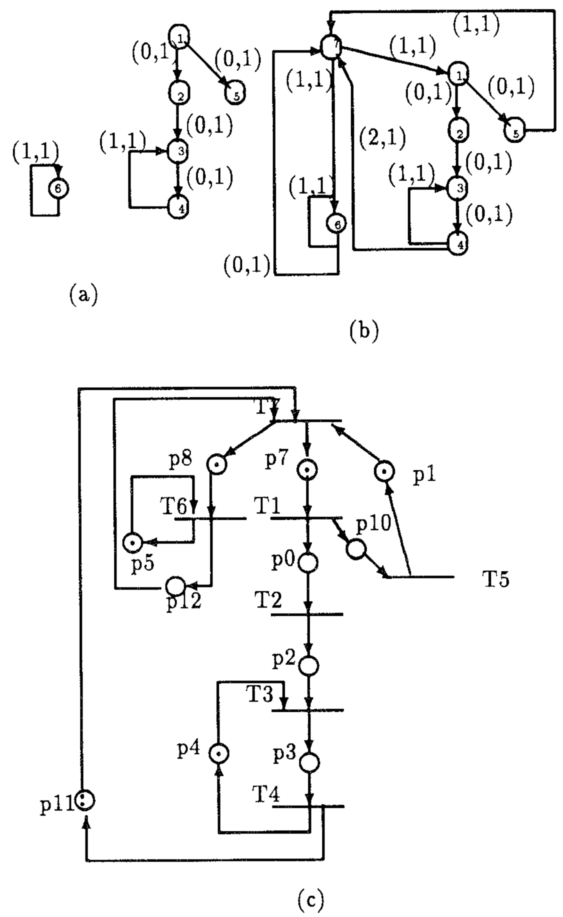
The firing of a transition can be thought of as passing the result of an operation performed at a node to other nodes that are waiting for this result. If the Petri net is cyclic, a state may be reached when a series of firings of the Petri net take it to a state through which it has already passed. The state of the Petri net is represented by the token count at each place. Because all decisions are deterministic, an identical state (with an identical reservation table) means the behavior of the Petri net repeats.

Min values of greater than 1 are handled by inserting dummy nodes so that no *min* is greater than 1. For example, an arc ($a \rightarrow b$, *dif*, 3) is replaced by arcs ($a \rightarrow t_1$, *dif*, 1), ($t_1 \rightarrow t_2$, 0, 1), and ($t_2 \rightarrow b$, 0, 1), where t_1 and t_2 are dummy nodes. This implementation increases the node count, but greatly simplifies the recognition of equivalent states as the marking contains all delay information. This algorithm handles *min* values of zero by doing a special firing check for nodes connected to predecessors by *min* times of zero. Thus compile time is increased by accommodating *min* times of zero.

Arcs are added to the DDG to make the graph strongly connected, eliminating the problem of leading chain synchronization. The benefit is that the rate of each firing is controlled; no node is allowed to sustain a rate faster than the slowest cycle dictates. As arcs are added, new cycles are created. If a new cycle θ' has a larger $min_{\theta'}/dif_{\theta'}$ than that contained in the original graph, the schedule is necessarily slowed down (II increased).

For simplicity, the number of added arcs is kept to a minimum. Formally, this is done as follows. If D is the DDG, let D' represent the acyclic condensation of D in which each strongly connected component of D is replaced by a single node. For a node $d' \in D'$, there exists a set of nodes $C_{d'} \subseteq D$ that correspond to d'. If there is a single source node[19] in D', let s' be that node. Let s be arbitrarily selected from $C_{s'}$. If there is more than one source node in D', create a set of nodes $R \subseteq D$ in which each source node of D' has a corresponding node. Let s be a dummy node added to D. For each $r \in R$, add an arc ($s \rightarrow r$, 1, 1) to the DDG. For

[19] A source node has no predecessors.



	Marked Places (tokens)	Schedule
0	4(1),5(1),7(1),8(1),9(2),10(1)	1, 6
1	0(1), 1(1), 4(1), 5(1),6(1), 9(2),10(1)	2, 5
2	2(1),4(1),5(1),7(1), 8(1),9(1),10(1)	3 1, 6
3	0(1),1(1),3(1),5(1),6(1),9(1),10(1)	4 2, 5
4	2(1), 4(1), 5(1), 7(1), 8(1),9(1),10(1)	

(d)

Figure 36. (a) DDG. (b) After being made strongly connected. (c) Petri net. (d) Schedule assuming infinite resources.

each node *k*′ ∈ *D*′ which is a sink,[20] arbitrarily select node $k \in C_{k'}$ and add an arc ($k \to s$, *dif*, 1). The values of *dif* are selected so that any cycle θ formed has a min_θ/dif_θ ratio as close as possible to the lower bound for *II* without exceeding it. For example, in Figure 36(b) the DDG of Figure 36(a) is made strongly connected. The acyclic condensation consists of nodes

[20] A sink has no successors.

(representing the sets of nodes from the original graph) {1}, {2}, {3, 4}, {5}, and {6}. The nodes of the acyclic condensation {1} and {6} are both sources. Thus a dummy node 7 is created as s and arcs (7 → 6, 1, 1) and (7 → 1, 1, 1) are added. The nodes of the acyclic condensation {6}, {3, 4}, and {5} are sinks. One node from each of the acyclic condensation sink nodes is connected to node 7. This results in adding arcs (6 → 7, 0, 1), (4 → 7, 2, 1), and (5 → 7, 1, 1) to the DDG. *Dif* values are selected to force newly formed cycles to have $min_\theta/dif_\theta \leq 2$.

Once the DDG has been modified so that it is strongly connected, the corresponding Petri net is created. Basically, nodes become transitions in the Petri net, each arc becomes a *pair* of arcs in the Petri net, and places are inserted between dependent transitions to keep track of what may fire next. The formal rules are as follows:

(1) For each node i in the DDG, a transition T_i is created.

(2) For each arc in the DDG from node i to node j, a place p is created along with arcs from T_i to p and from p to T_j.

At each point in time, the transitions that fire form a parallel instruction of the schedule. When the marking repeats, a software pipeline exists. The schedule repeats whenever the Petri net enters a state (defined by the placement of tokens) in which it has been before. For a good discussion of the properties of cyclic Petri nets, see Gao et al. [1991b].

The initial placement of tokens must be specified. Initially, any node that is dependent on a previous iteration is ready to be executed (as there is no previous iteration to deny execution). Thus one would guess that places that model arcs representing loop-carried dependences would need to be marked. The question comes in deciding how many tokens should be placed there. The number of tokens necessary depends on the *dif* value for that arc. A *dif* of zero corresponds to *no* tokens. Let b^i represent operation b from the i^{th} iteration. A *dif* of three on arc $a \rightarrow b$ indicates that b can get three iterations ahead of a. Thus b^4 must follow a^1; yet, b^1, b^2, and b^3 are unconstrained. Therefore, the number of tokens at a place is just the *dif* of the corresponding arc in the dependence graph. The formal rule follows.

(3) The initial marking of the Petri net is such that for an arc from i to j in the DDG with *dif* value d, the corresponding place p has d tokens assigned to it. This allows node T_j to be fired up to d times before node T_i is fired.

Modeling Resources. Any restriction (other than dependence) is modeled as a resource constraint. If we had a single adder, operations needing the adder could not be scheduled together. Resource usage that is regular or nonpersistent can be handled by introducing resource places. The rules governing resource places are as follows:

(4) For each resource, a place p_r is created. The place is assigned the same number of tokens as the number of instances of that particular resource.

(5) Resource usage is controlled by requiring that all nodes needing the resource be cyclically connected to this place. If a node uses a particular resource, then there is a loop from that transition to the resource place and back. Because a node needs a token on each of its input places before it can fire, an arc from a resource place requires that the resource is available before scheduling the node. The arc to the resource place allows the node to return the resource after use.[21]

[21] If an operation required more than one instance of a given resource, the arcs could be weighted with the number of tokens required from that source

Modeling persistent irregular resources requires a more complicated model. Instead of using resource places (that can only ensure a resource is available at a given point in time, not future availability) a current *reservation state* must be kept. A reservation state (RS) is the union of all reservation tables (offset to reflect starting time) for operations that have begun execution but have not completed. Consider an operation a that is to be scheduled. If the corresponding reservation table R_a has length $|R_a|$, each row must be intersected (binary *and*) with the corresponding row of the reservation state. If any resulting row is nonzero, there is a resource conflict and the operation cannot be scheduled. If there are no conflicts, each row is unioned (binary *or*) with the corresponding row of the reservation state to reflect the current and future resource needs. After all operations for the current time cycle have been scheduled, the reservation state is advanced to reflect the passage of time, that is, $\mathrm{RS}[i] = \mathrm{RS}[i + 1]\ \forall i = 0 \ldots |RS| - 1$.

Behavior Table. The schedule shown in Figure 36(d) is termed the *behavior table.* Row 0 of Figure 36(d) indicates the initial marking of each place in the Petri net; the number of tokens at each place is shown in parentheses. The column marked "Schedule" indicates the transitions that fire given the current marking. For simplicity of presentation, it is assumed there are no resource conflicts. As the state at instruction 2 and the state at instruction 4 are identical, the kernel starts at instruction 2 and ends at instruction 3. The kernel with an initiation interval of two is marked with a box. In order to incorporate persistent irregular resource usage, each row of the behavior table has an attached resource state describing the current and future resource commitments. For simplicity, the attached resource state (dummy transition 7) is not shown.

This algorithm is enhanced by the addition of a pacemaker that regulates the greediness of the algorithm [Allan et al. 1993]. A *pacemaker* is a cycle of dummy nodes added to the Petri net that has a min_θ/dif_θ equal to the estimated *II*. The pacemaker passes tokens to the rest of the Petri net to control the rate of firing. The pacemaker attempts to force the estimated pace so a repeating section is formed quickly. In the example of Figure 36(b), a pacemaker to ensure that nodes do not fire more frequently than every two cycles is not required as the schedule exhibits that property already. In larger examples, a pacemaker can reduce the compile time to obtain the schedule, the *Span* for the kernel, and the achieved initiation interval.

This general model for software pipelining has the advantage of being easily extendible. For example, predicates within the loop body are handled by passing the token along paths representing both the branches and by having a merge node that does not fire until it receives a token from both branches. The control dependence itself is treated as a form of data dependence. The Petri Net Pacemaker (PNP) technique performs optimizations such as renaming and forward substitution to improve the performance of loops containing predicates [Rajagopalan and Allan 1994].

The Petri net approach solves the problem that Aiken and Nicolau [1988c] and Zaky [1989] faced of having two parts of the graph execute at different rates. Making the graph strongly connected forces all transitions to fire at the same rate. The number of tokens at each resource place reflects the number of resources of each type. Thus when a transition fires, all other transitions competing for the same resource are prohibited from firing as the token is not available. To maintain determinism, the selection algorithm must be consistent in choosing which candidates fire when there are competing choices. The priorities used in scheduling are similar to other scheduling algorithms. Each transition is augmented with an execution count that is incremented each time the transition fires. When several transitions are able to fire, the selection algorithm is able to fire the transition that has fired least and hence is from the earliest iteration. Transitions are also prioritized so that within the same iteration, transitions that are part of critical cycles are selected first.

The Petri net approach also solves the problem of persistent irregular resources. The procedure to identify an identical state is broken down into steps. Step one uses hashing on the encoding of the tokens at each place to discover when the tokens are in an identical configuration. For each state on the matching hash list, step two checks to see if the current global reservation table agrees with the global reservation table at the previous step. For a sufficiently large hash table, the number of entries to check is small (one or two) and the test for an identical state only proceeds until a single element differs, so the check for identical state requires minimal effort.

Comparison With Other Methods. PNP is similar in spirit to Perfect Pipelining [Aiken and Nicolau 1988a, 1988b; Nicolau and Potasman 1990]. A problem for Perfect Pipelining is the difficulty of determining when nodes are functionally equivalent (i.e., the pattern is repeating). Because the Petri net uses a combination of places and global reservation tables to record state information, a repeating pattern can reliably be determined. Perfect Pipelining also creates kernels in which duplicate operations are present even when doing so does not improve performance. For example, a loop length of 4 with two copies of each operation (for an effective initiation interval of 2) is generated when a loop length of 2 (with one copy of each operation) is equivalent. This results from the problem of scheduling each node greedily. Because each node is allowed to execute as fast as possible as long as gaps do not occur in the schedule (preventing a pattern from forming), some operations are scheduled more frequently than the effective initiation interval. Later, other operations have to be delayed to satisfy dependence constraints. Inasmuch as the pacemaker regulates the rate of all operations, such rate fluctuations (that increase the code size) are less likely. For a thorough discussion of these problems see Jones and Allan [1990].

The PNP algorithm is compared with Lam's algorithm on loops with both low and high resource conflicts [Rajagopalan and Allan 1994]. With low resource conflicts, the PNP algorithm does marginally better. With high resource conflicts, the PNP algorithm shows a significant improvement of 9.2% over Lam's algorithm. PNP often finds a lower *II* as fractional rates are required. Because Lam's algorithm schedules each strongly connected component separately, a fixed ordering of the nodes belonging to each strongly connected component in the schedule is used. This fixed ordering, together with the resource conflicts between the nodes of the strongly connected component and other nodes of the schedule, restricts the overlap between them. This forces Lam to use a higher value of *II* which in many cases is not optimal.

The PNP algorithm is compared with Vegdahl's [1992] technique that performs an exhaustive search of all the possible schedules to look for a schedule with the shortest length. PNP compares quite favorably with this exhaustive technique. The compile time of PNP is negligible compared to Vegdahl's exhaustive technique.

3.3 Vegdahl's Technique

Unlike all other techniques discussed in this survey, Vegdahl's represents an exhaustive method in which all possible solutions are represented and the best is selected. As software pipelining is NP-complete, this makes the algorithm impractical for real code.

Vegdahl [1992, 1982] uses an exhaustive technique that relies on a decision graph of choices. The method uses a dynamic programming scheduling algorithm to schedule straight line code formed from the loop body by unrolling. Vegdahl builds an *operation set graph* in which nodes represent the set of opera-

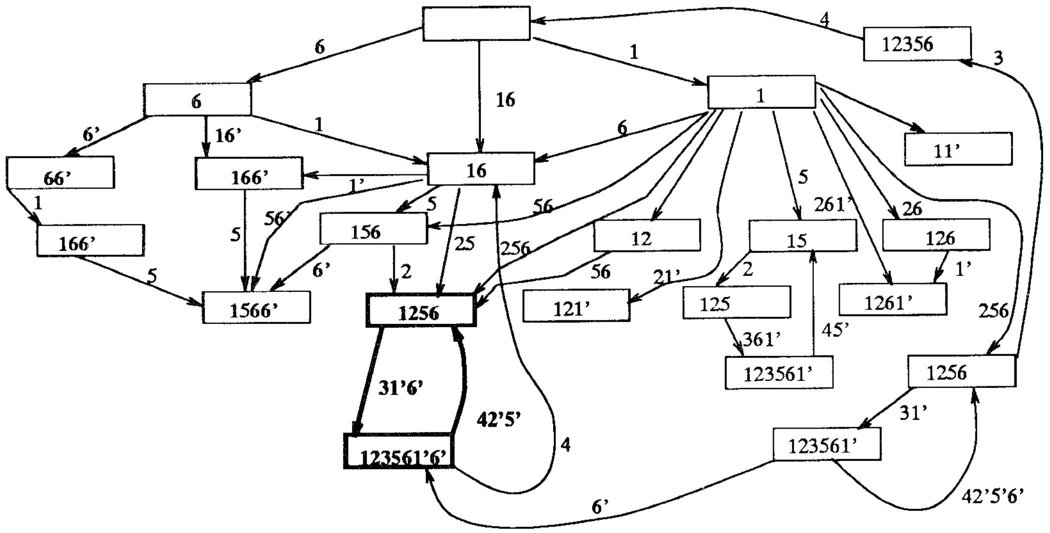


Figure 37. Partial operation set graph for the example of Figure 28.

tions scheduled so far and arcs represent a series of scheduling decisions. In essence, the nodes represent state information similar in usage to the markings in the Petri net algorithm, the difference being that the operation set graph represents *all* possible scheduling decisions at every point in time rather than a single decision. Although the concept is like a decision tree [Smith 1987], the diagram becomes a graph because nodes representing equivalent states are merged. Rather than considering loop-carried antidependences directly, Vegdahl defines "duals" of the dependences to represent antidependences. This appears to be an idiosyncrasy of the data representation rather than an inherent part of the concept. The automatic interpretation of dependence arcs is a disadvantage if register renaming eliminates loop-carried antidependences.

Consider the partial operation set graph for the example of Figure 37. Because the graph is so large, only representative parts of it are shown. Inasmuch as the span of the schedule obtained by PNP is two, it is reasonable to apply Vegdahl's exhaustive technique to a loop body that has been unrolled twice (two copies exist) as this allows operations from adjacent iterations to be scheduled together. Initially no operations have been scheduled (represented by the initial node with no label). There are three choices: 1 can be scheduled, 6 can be scheduled, or both 1 and 6 can be scheduled. These three choices are represented by three arcs emanating from the initial node. Suppose 6 is scheduled first, then there are three choices for the next time step: 1 can be scheduled, operation 6 from the next iteration (6′) can be scheduled, both 1 and 6′ can be scheduled. These choices are represented by arcs coming out of the node labeled 6. Note that the node labeled 6 indicates operation 6 has been scheduled, and an arc labeled 6 indicates 6 is to be scheduled at this time step. If 6 is scheduled in the first time step and then 1 is scheduled in the second time step, the node 16 is reached. Notice this is the same node reached if operations 1 and 6 were scheduled initially. The scheduling process is as follows:

(1) Create an empty initial node and mark it as open.

(2) While there exists an open node, select an open node *curr*.

    (a) While observing both resource conflicts and dependence constraints, consider all possible operations that can be scheduled given the operations of *curr* have executed. Let *exec* be one possible set of operations that can be executed. Let *next* represent the new state formed from executing *exec* when *curr* was the previous state. If *next* contains every operation from one iteration, *reduction* is applied by removing the set of operations and shifting the remaining operations by one iteration, for example, x' becomes x, x'' becomes x'. (In our example, when we execute 42′5′ at node 123561′6′ this results in node 1256 rather than 1234561′2′5′6′.) If *next* is already in existence as a node, create an arc from *curr* to that node with label *exec*. Otherwise, create a new node representing *next* and create an arc from *curr* to *next* with label *exec* and mark the new node as open.

(b) Repeat the preceding step for each possible set *exec*.

(3) Once there are no more open nodes, locate all cycles in the graph. Cycles of minimum length are possibilities for the kernel of the software pipeline.

Kernel recognition is handled by the merging of equivalent nodes followed by finding minimal cycles. In our example, one cycle (shown in bold) consists of instructions 31′6′ and 42′5′ and represents the same kernel found in Figure 36(d). Another possible schedule is shown by the schedule 31′ and 42′5′6′. A nonoptimal schedule is indicated by 2, 361′, and 45′. Numerous other possibilities exist but have been omitted from the operation set graph in the interest of readability. This method does not generate prelude or postlude, although this code can easily be generated given the kernel. In order to evaluate which cycle of minimal length is best, prelude and postlude code must be considered.

Because the algorithm is exhaustive, it produces optimal results but is obviously inappropriate for graphs of any size. The algorithm works best when the graph is tightly constrained and has many resource conflicts as fewer state nodes in the operation set graph are possible. This algorithm has many similarities to the Petri net algorithm. *Min* values of greater than one are implemented by introducing dummy nodes. Vegdahl [1992] states that *dif* values of greater than one can be handled, but this has not been fully tested. Although the use of dummy nodes addresses the problem of nonunit latencies, persistent resource usage cannot be modeled. Although Vegdahl never mentions fractional rates, this method clearly has the potential of achieving fractional initiation intervals. Vegdahl's algorithm differs from the Petri net algorithm in its exponential time complexity as well as in the fact that it physically unrolls the loop a "sufficient" number of times before the algorithm begins. An upper bound on the number of nodes in the operation set graph is the cardinality of the power set[22] of the set containing all operations of the unrolled loop. Because the size of the unrolled loop may have such a negative effect on the complexity, it is important to be able to predict the amount of unrolling necessary to achieve an optimal schedule. Many of these nodes are never present in the operation set graph as they do not represent a legal state in the firing sequence constrained by dependences and resource conflicts.

Predicates have not been implemented but the author alludes to the possibility that conditionals could be handled by combining the algorithm with trace scheduling.

[22] A power set of set P is the set of all subsets of P. For a set P having p elements, the power set has 2^p members.

Although this algorithm is impractical because of its complexity, it may be useful for very small examples, and it underscores the reason that researchers look for heuristics.

4. ENHANCED PIPELINE SCHEDULING

Enhanced Pipeline Scheduling integrates code transformation with scheduling to produce excellent results. One important benefit is the ability to schedule various branches of a loop with different *II*.

Ebcioğlu and Nakatani [1990, 1987] propose an algorithm unlike either modulo scheduling or kernel recognition algorithms. Enhanced Pipeline Scheduling uses a completely different approach to software pipelining, building on code motion (like Perfect Pipelining), but retaining the loop structure so no kernel recognition is required. This algorithm is quite complicated both to understand and in the code produced. However, it has benefits no other algorithm achieves as it combines code transformation with software pipelining.

Code motion pipelining can be described as the process of shifting operations forward in the execution schedule. Because operations are shifted through the loop control predicate, they move ahead of the loop, into the loop prelude *and* they move back into the bottom of the loop, as operations from a later iteration. The degree to which shifting is able to compact the loop depends on the resources available and the data dependence structure of the original loop. The algorithm has some similarity to Perfect Pipelining in that it uses a modification of Perfect Pipelining's global code motion algorithm, but differs significantly because Enhanced Pipeline Scheduling manipulates the original flow graph to create $\mathscr{K}$ and does not require searching for a pattern in unrolled code. Because the loop is manipulated as a whole rather than as individual iterations, Enhanced Pipeline Scheduling does not encounter the problems of pattern identification inherent in Perfect Pipelining.

Software pipelining occurs when operations are moved across the back edge,[23] allowing them to enter the loop body as operations from future iterations. The pipelining algorithm iterates until all the loop-carried dependences have been considered. The pipeline prelude and postlude are generated by the automatic code duplication that accompanies the global code motion.

This model can handle *min* times of greater than 1 by inserting dummy nodes. However, the algorithm cannot handle persistent irregular resources.

4.1 Instruction Model

Enhanced Pipeline Scheduling is a powerful algorithm that can utilize a multiway branching architecture with conditional execution. The conditional execution feature makes this machine model more powerful than the original Perfect Pipelining machine model that includes only multiway branching. Recent versions of Perfect Pipelining have included multipredicated execution [Nicolau and Potasman 1990]. In order to take advantage of multipredicated execution, a more powerful software pipelining algorithm is required.

Figure 38 shows a typical control flow graph node,[24] I_1, termed a *tree instruction*. I_2 through I_4 are labels of other control flow graph nodes. The condition codes $cc1$ and $cc2$ are computed *before* the node is entered, and only those statements along one path from the root to a leaf are executed. All the operations on the selected path are executed concurrently, using the old values for all operands. For example, in Figure 38 if $cc1$ is *true* (left branch) and $cc2$ is *false* when I_1 is executed, operations O_1, O_3, O_7, and O_8 are executed simultaneously,

[23] A back edge of a data dependence graph is an edge whose head dominates its tail in the flow graph and is used to locate a natural loop [Aho et al. 1988]. A node a dominates a node b in a graph if every path from the source to b must contain a

[24] A Control Flow Graph (CFG) is a graph in which nodes represent computations and edges represent the flow of control [Aho et al. 1988].

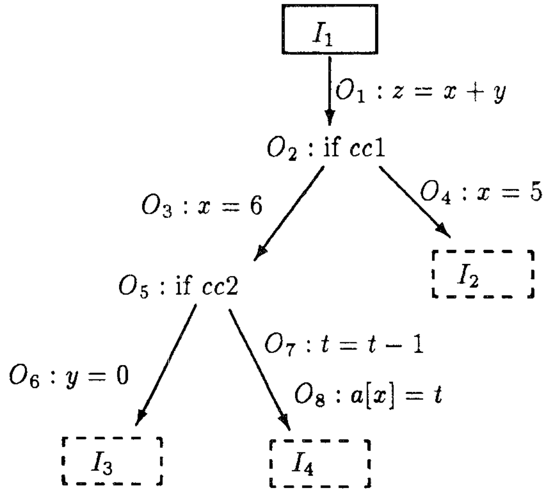


Figure 38. Tree instruction used ın enhanced pıpeline scheduling.

then control is transferred to I_4. Thus the assignment to t by O_7 does not effect the use of t by O_8. Two types of resource constraints are placed on this machine model. One is limited by the total number of operations that can be contained within the tree instruction. One is also limited to a fixed number of different paths through the tree instruction.

With conditional execution, operations can be placed on any branch of a CFG node, unlike a traditional machine model in which all operations must precede any conditional branch operations. Although Enhanced Pipeline Scheduling can utilize such features, the scheduling technique does not require this powerful architecture. Other architectures can be represented by the tree instruction by restricting the form of the instruction. For example, architectures that do not support multiway branching or conditional execution can be modeled by tree instructions.

4.2 Global Code Motion With Renaming and Forward Substitution

Nakatani and Ebcioğlu [1989] use renaming and forward substitution[25] to move operations past predicates and shorten dependence chains involving antidependences. *Renaming* involves replacing an operation such as $x = y\,op\,z$ with two operations $x' = y\,op\,z; x = x'$. Because $x' = y\,op\,z$ defines a variable that is only used by the copy statement $x = x'$, the assignment to x is free to move outside the predicate. When $x' = y\,op\,z$ moves before the predicate that controls its execution, the statement is said to be *speculative* because the statement is only useful if the branch is taken. Figure 39 illustrates the shortening of a data dependence chain (involving an antidependence) and a control dependence chain using renaming of a. Figure 39(a) shows the original dependence graph, whereas Figure 39(b) shows the graph after renaming. The dependence between n_5 and n_4 can be eliminated by forward substitution. For an assignment statement *var* = *expr*, when uses of *var* in subsequent instructions are replaced by *expr*, it is termed *forward substitution.* This is particularly useful if, as a result of the forward substitution, the operations can be executed simultaneously. In Figure 39(c) forward substitution is used to change $d = 2 * a$ to $d = 2 * a'$ because the a referenced has the value of a'.

[25] The authors term this combining.

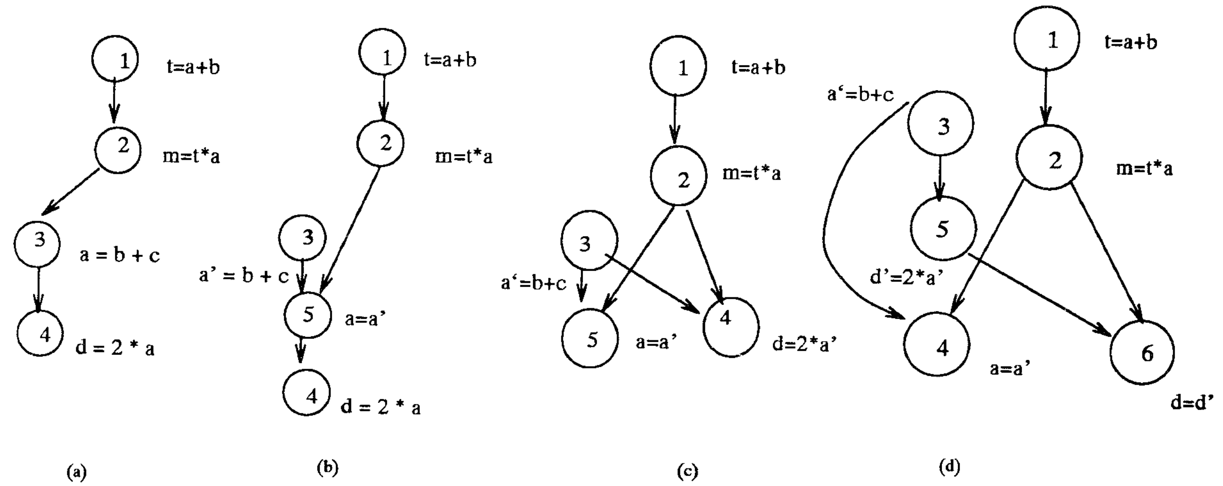


Figure 39. (a) Original dependence graph. (b) Dependence graph after renaming $a = b + c$. (c) After forward substitution. (d) Dependence graph after renaming $d = 2 * a$

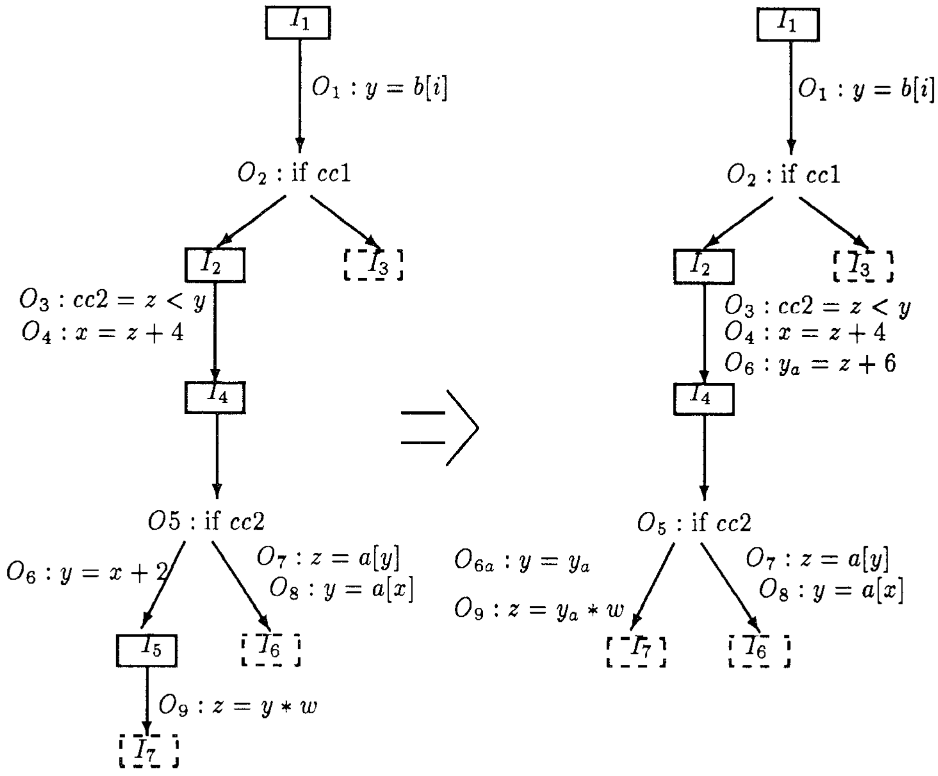


Figure 40. Example of code motion of O_6 with renaming and forward substitution of $x = z + 4$ for the x in O_{6a}

Figure 39(a) shows that $d = 2 * a$ is renamed. Notice that in this case the length of the dependence chain for the graph is decreased.

To see the flexibility of this model, node 2 could also represent a predicate and the same transformations used on the data dependences also can be applied to the control dependences.

Forward substitution may collapse true dependences that prevent code motion. A true dependence between two operations O_1 and O_2 may be collapsible with forward substitution if (1) O_1 is a copy operation, or (2) both operations involve a constant immediate operand. Forward substitution is performed if a true dependence can be collapsed.

Figure 40 shows an example in which both forward substitution and renaming are required to move an operation. The diagram shows part of a CFG before and after operation O_6 is moved from instruction I_4 to instruction I_2. Note that multiple tree instructions are shown in the figure. Each rectangular node is the entry point of a tree instruction. Dashed boxes represent branches to the tree instructions indicated. O_6 has a true dependence on O_4 that involves an immediate operand, making this dependence a candidate for forward substitution. First, y is renamed giving $O_6 : y_a = z + 6$ and $O_{6a} : y = y_a$. Next, forward substitution is performed giving $O_6 : y = z + 4 + 2$. Constant folding is then performed giving $O_6 : y = z + 6$. To see the advantage of this renaming, O_9 is moved from I_5 eliminating I_5. This motion requires forward substitution as $O_9 : z = y_a * w$. Notice that the total execution time of the code has been reduced.

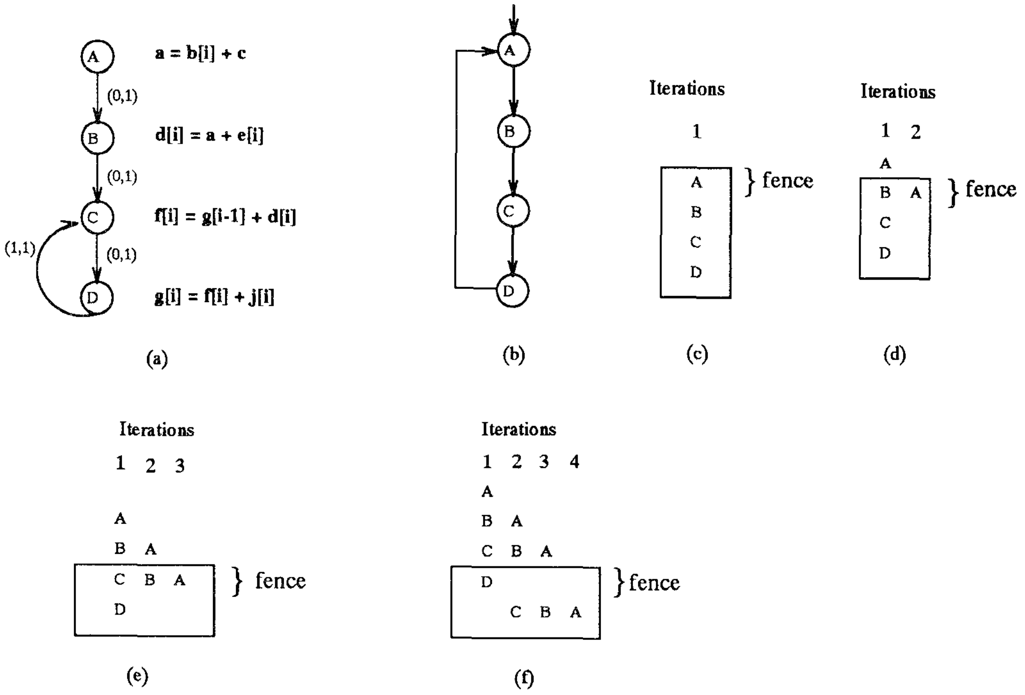


Figure 41. Enhanced pipeline scheduling. (a) DDG of loop. (b) Control flow graph of loop. (c) After filling first instruction. (d) After filling second instruction. (e) After filling third instruction. (f) Final software pipeline.

4.3 Pipelining the Loop

To perform pipelining, code is moved backward along the control flow arcs. The algorithm consists of two phases that repeat until all operations have an opportunity to move. In phase one, code from within the current loop body is moved as early as dependences allow. The first parallel instruction(s) of the loop is termed a *fence*. The name is derived from the fact that code is moved from the rest of the loop to a position as early as possible in the loop, but not past the fence. Hence, the fence bounds the code motion. In phase two, the fence instruction is duplicated and moved. Code from the loop body is duplicated as it moves past the top of the loop because there are *two* control predecessors. The code that moved out of the loop forms a prelude. The code that joins code at the bottom of the loop body represents work originally performed in a different iteration. This process continues until all statements from the original loop body have an opportunity to move [Ebcioğlu and Nakatani 1990].

For example, consider the example of Figure 41. The four statements of the loop are connected by true dependences. Thus when the fence instruction contains operation *A*, no other operations from the loop can move up as shown in Figure 41(c). Once a fence instruction is filled, it is moved out of the (top of the) loop, where it becomes a part of the loop's prelude. A fence is considered filled if all resources are consumed (or nearly consumed) or if all code motion to the fence has been exhausted. In early versions of the algorithm, code migrated upward before it was known whether the code could even be executed early. This wasted motion (which also caused race conditions) was eliminated in recent versions. The filled fence is also duplicated to follow the loop back edge of the CFG where it rejoins the loop (at the bottom). In Figure 41(b) the back edge is shown from D to A and represents the fact that A is executed after D. This motion across the back edge moves operations between iterations. Hence software pipelining is achieved.

Nonunit latencies are handled by inserting dummy nodes, but there is no provision for persistence resource constraints.

In Figure 41(d) the fence instruction from Figure 41(c) (consisting of operation A) is moved into the prelude and back into the loop as can be seen by the two copies of A. Because the A that moves back into the loop is from the second iteration, it is not dependent on any operations from the first iteration. Thus operation A moves all the way to the new fence instruction. In the next stage (Figure 41(e)) the fence of Figure 41(d) consisting of BA is moved into the prelude and back into the loop. As neither B nor A (from the second and third iterations, respectively) depend on operations from the first iteration, they move to the top of the loop. In the last step (Figure 41(f)) the fence CBA is moved into the prelude and back into the loop as operations from iteration two, three, and four, respectively. In this case, the C from iteration two is dependent on the D from iteration one so these operations do not move into the fence. Because all operations have been considered in a fence, the pipelining algorithm terminates.

In general, operations from previous fence instructions move all the way through to the current fence (when data dependences permit), allowing operations from many iterations to move as the fence is moved across the back edge. This process continues until all instructions in the original loop body have moved up to the top for their chance as a fence, or when all data dependences in the loop have been seen. Note that at every stage in the process a valid loop is maintained.

Enhanced Pipeline Scheduling uses a CFG and the dynamic data dependence information of the loop body. Each node in the CFG represents a tree instruction containing several operations that are executed concurrently. Initially all the instructions contain a single operation. Figures 42–45 illustrate the process on a sample loop from Ebcioğlu and Nakatani [1990]. In Figure 42(a) each operation is shown as a separate instruction. During Enhanced Pipeline Scheduling, code motion is always toward the top of the loop. The instruction(s) that has no predecessors in the loop independent subgraph is termed the fence and is shown in the figure as a dotted box.

Figure 42(b) shows the loop after all possible operations have moved out of the false (right) branch of the $!cc1$ conditional in I_5 and upward as far as possible. By convention, we always move operations from the false branch before considering possible code motion in the true branch.[26] In Figure 42(b) note that the operation $x = g(x)$ must be renamed as it moves past the branch on $!cc1$ as x is live[27] on the true branch. During the upward migration, the movement of operations from all successors of a node must be attempted before code motion is allowed from the node. For example, if both I_6 and I_7 have an identical operation, performing code motion on successors of I_5 allows such duplicates to be recognized and merged. Otherwise, the two copies might move up to create redundant code. Because duplicate copies are often created by the code motion algorithm, this becomes important. The merging, termed *unification*, of identical operations at branch points of the CFG is a key feature of both Perfect Pipelining and Enhanced Pipeline Scheduling that distinguishes their global code motion

[26] The order of code motions in the figures does not always follow the strict order of the algorithm. The result is the same,.and this order aids the presentation. By the strict sense of the algorithm, all possible migrations from I_5 would be attempted before migration is allowed from instructions located above it.

[27] A variable is live if it is used on some path before it is redefined.

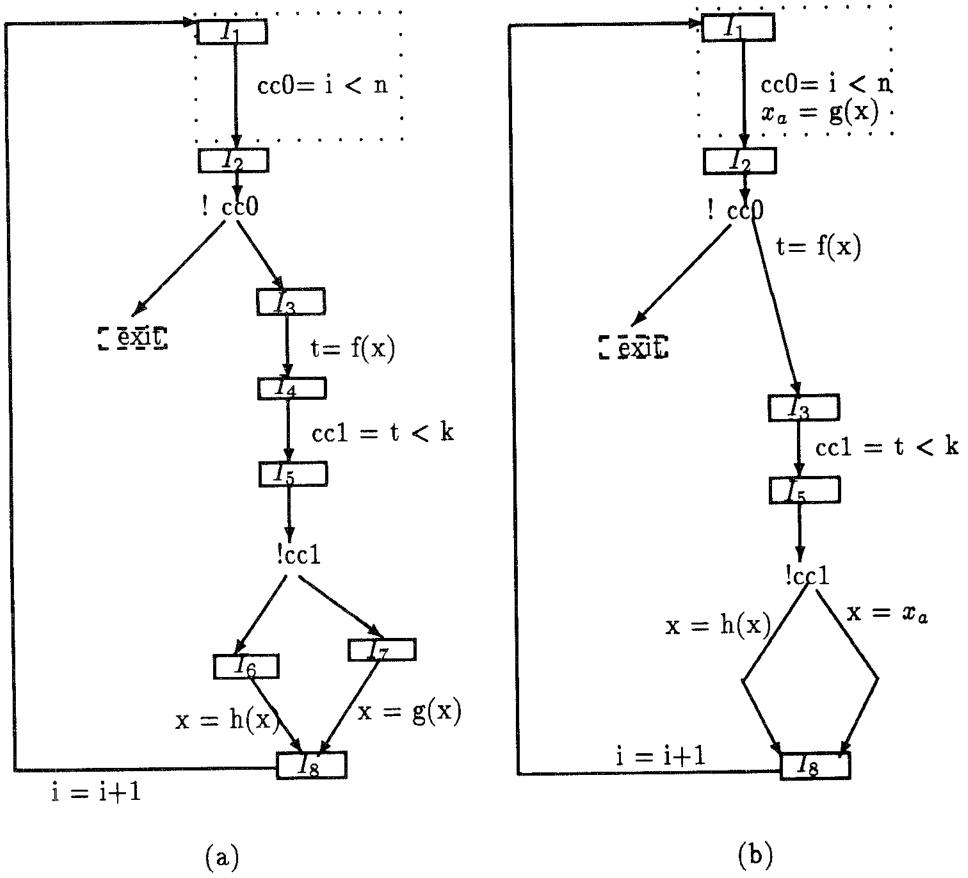


Figure 42. (a) Unpipelined loop. Dotted box represents the fence. (b) Loop after migrating from false branch of the !*cc*1 conditional.

from other global code motion algorithms [Fisher 1981].[28]

Figure 43(a) shows the loop after the fence instruction I_1 has been filled. Note that when the operation $x = h(x)$ moves past the !*cc*1 branch, it does not need to be renamed as x is dead[29] on the other branch, but when it moves past the !*cc*0 branch, it must be renamed as x is live on loop exit.

Upward code migration stops when code movement possibilities encountered in that pass have been exhausted. Thus at Figure 43(a) code motion stops as no further movement exists. Any empty instructions are removed from the CFG; that is why I_4 is eliminated. Note that removing empty instructions causes no problems in adhering to required *min* times. Because necessary delays are achieved by the insertion of dummy nodes, dummy nodes within an instruction prevent the instruction from being eliminated. Persistent resources cannot be handled with this model. Both the insertion and removal of empty instructions cause problems in the strict timing constraints of consecutive instructions imposed by persistent resources.

After code motion stops, the current fence is marked as *filled* and all immedi-

[28] These features have been added to a later version of trace scheduling [Freudenberger et al. 1994].

[29] A variable is dead if it is not used before it is defined on all branches.

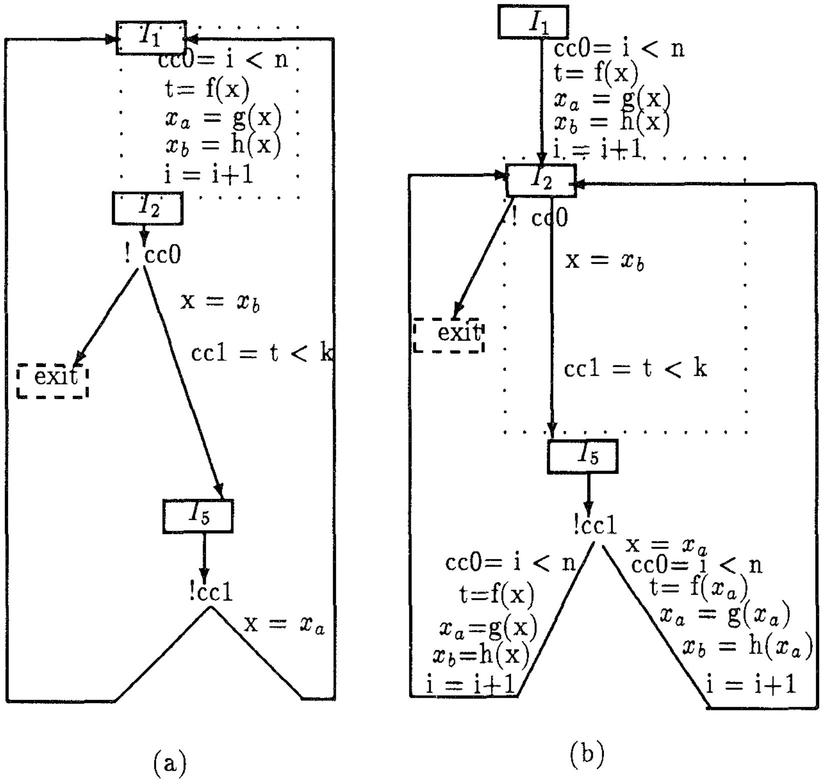


Figure 43. (a) Loop after filling fence I_1 (b) Loop after moving filled fence I_1 to bottom of loop.

ate successors of the current fence become the new fence instructions. Figure 43(b) shows the filled fence instruction migrating upward past the join point of the loop in which the program enters the loop and the loop back edge returns to the top of the loop. When this motion takes place, the instruction enters the pipeline prelude and also enters the loop as the first instruction from iteration two (I_1^2). In Figure 43(b) the old fence instruction is not seen at the bottom of the loop because all the operations it contained moved into I_5. Note the forward substitution of x_a for all uses of x in operations from the old fence as they move into the false branch of the $!cc1$ conditional. No analogous transformation is required on the true branch as it is empty.

Moving the filled fence instruction creates at least two copies of the fence: one on the path entering the loop and one at the end of the loop. In this example there are three copies produced because there are three predecessors of I_1.

As these operations move up, a software pipeline is created. The new fence instructions are filled by the same migration process. Enhanced Pipelining Scheduling repeatedly fills fence instructions until the loop contains only one instruction, or the loop prelude and $\mathcal{K}$ contain operations from a sufficient number of iterations to see all the loop-carried dependences.

Figure 44(a) shows the loop after all possible operations have moved out of the false branch of the $!cc1$ conditional and upward as far as possible. The operations that resulted in unification have also been moved at this point. Note that the copies of operations $i = i + 1$ and $cc0 = i < n$ that appeared on both

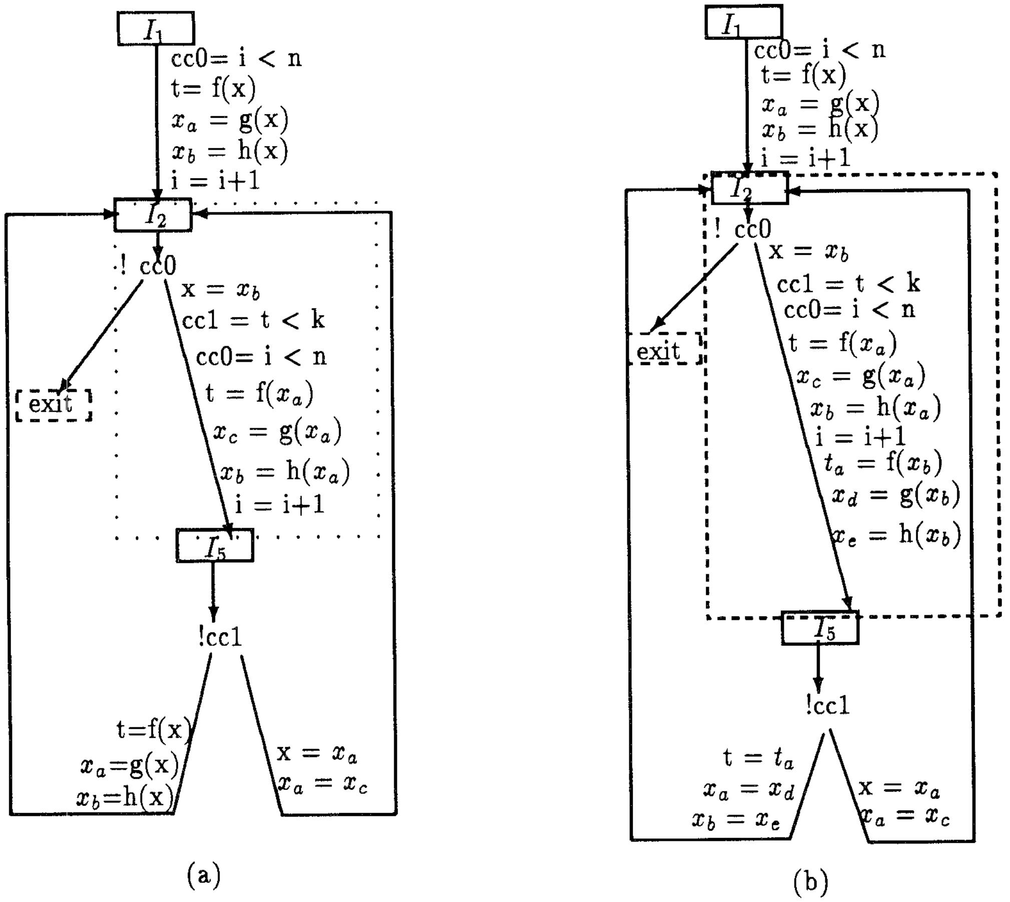


Figure 44. (a) Loop after migrating from false branch of the !*cc*1 conditional. (b) Loop after filling fence I_2.

branches of the condition !*cc*1, are each merged into a single operation as they move past the branch point. Note also that x_a has been renamed x_c as it moves past the use of x_a that is assigned to x.

Figure 44(b) shows the loop after the fence instruction I_2 has been filled. Figure 45 shows the final pipelined loop. $\mathcal{K}$ in this example is only one tree instruction. Enhanced Pipeline Scheduling starts with a large loop and reduces it in size as the pipelining proceeds. As the loop is always valid, the algorithm only iterates until the loop does not shrink in size. This occurs when the loop contains only one instruction or when all loop-carried dependences have been seen.

The copy operations to x appear to be redundant in $\mathcal{K}$ as x is never used. The copies to x are needed because x is live on exit from the loop. Note that the operations originally assigned to x are now executed speculatively and their results placed in renamed copies of x. The copy operations to x ensure the correct value is assigned to x on loop termination. The resulting loop is the pipelined loop body, and the prelude and postlude are generated automatically by the code duplication that occurs when operations are

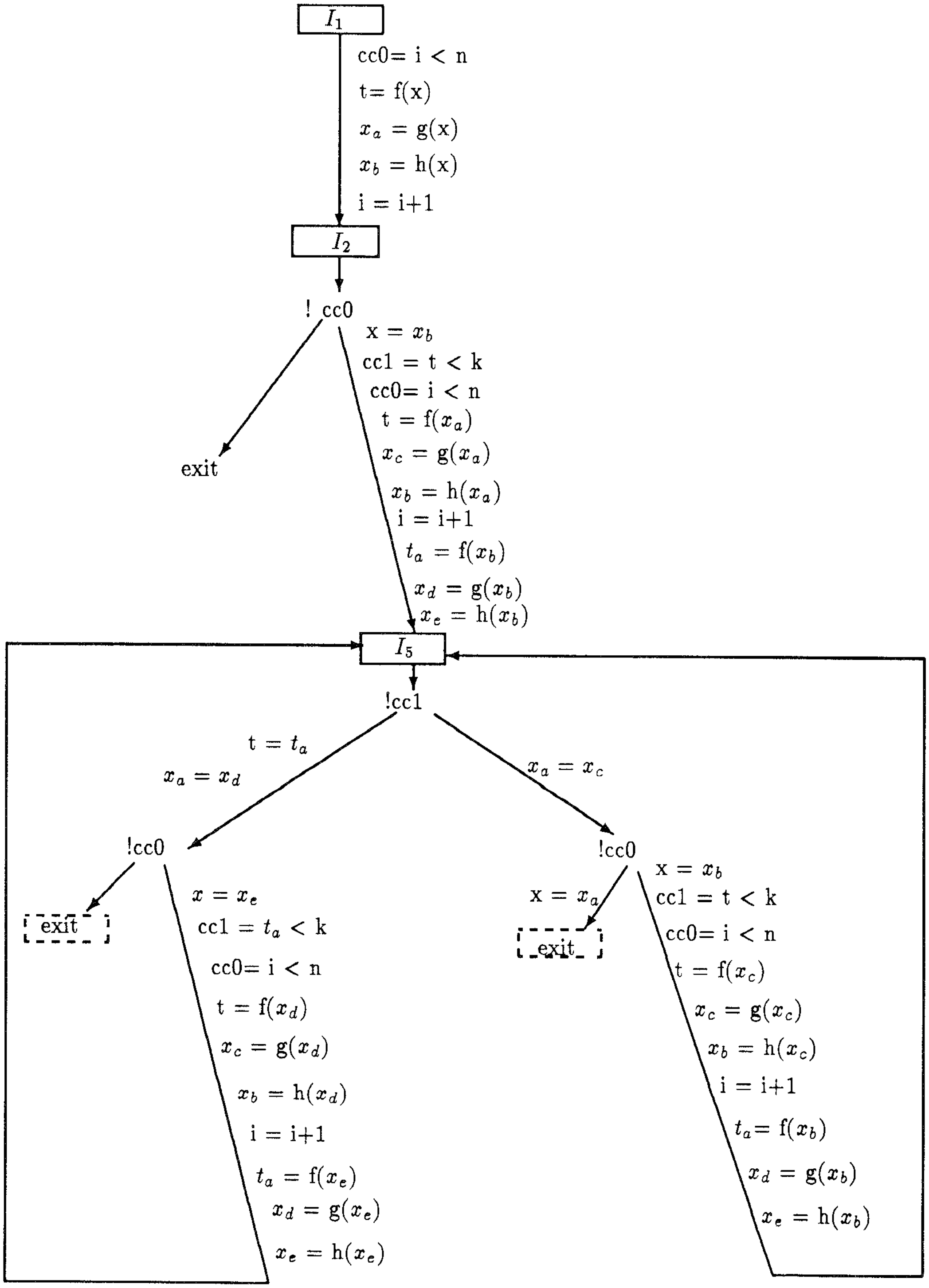


Figure 45. Final pipelined loop.

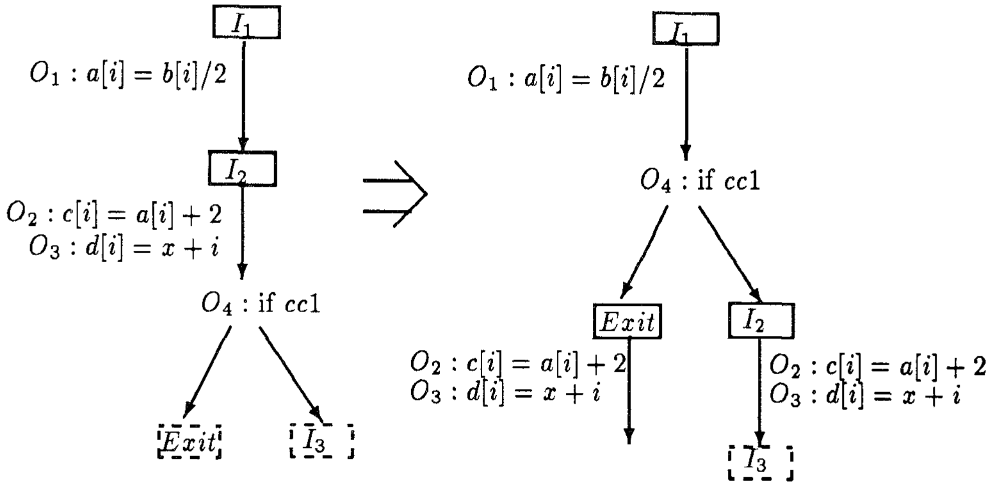


Figure 46. Example of copying operations to both branches of a moved branch operation.

moved in the CFG. This pipeline has no postlude because the exit branch never moves past other operations of the loop.

The postlude is generated when the conditional branch operation of the exit test is moved upward past code in the loop. When the branch is moved, any operations it moves past must appear on both branches, resulting in the loop postlude. For example, Figure 46 shows part of a CFG before and after the branch operation O_4 is moved to I_1. Note that O_2 and O_3 must be copied to the exit to become part of the postlude branch as they are executed regardless of the outcome of the conditional branch in the original loop. The pipeline postlude with Enhanced Pipeline Scheduling is typically small due to the renaming.

4.4 Reducing Code Expansion

Enhanced Pipeline Scheduling places a restriction on the movement of operations out of a filled fence to reduce the code expansion that occurs because of the copying of operations to both predecessors of a node. Operations are allowed to move out of a filled fence instruction only if all the operations can move, but once the operations have moved out of a filled fence instruction, they are free to move independently of each other. Without this requirement, given a node with two predecessors, it is possible that code motion would generate two versions of the node without any execution time benefit. However, this restriction can result in accepting a local minimum rather than achieving the global minimum. Enhanced Pipeline Scheduling does not have an optimal *II* value that it is trying to achieve, but stops when no further improvements are seen.

Figure 47 shows an example of this code duplication. When moving operations from I_3, O_4 can move to I_2 when forward substitution is used to collapse the true dependence, but O_4 cannot move to the other predecessor of I_3. The operation O_5 cannot move to either predecessor of I_3. If code motion is allowed in which operations of I_3 are allowed to move independently, two copies of I_3 are needed, one containing both O_4 and O_5, and the other containing only O_5. If I_3 is a filled fence, this type of code duplication would be prevented, as both O_4 and O_5 would have to move out of I_3 for any movement to occur. Code duplication is expensive, therefore Enhanced Pipeline Scheduling tries to reduce it.

In an improvement of Enhanced Pipeline Scheduling, Nakatani and Ebcioğlu [1990] introduce a windowing technique

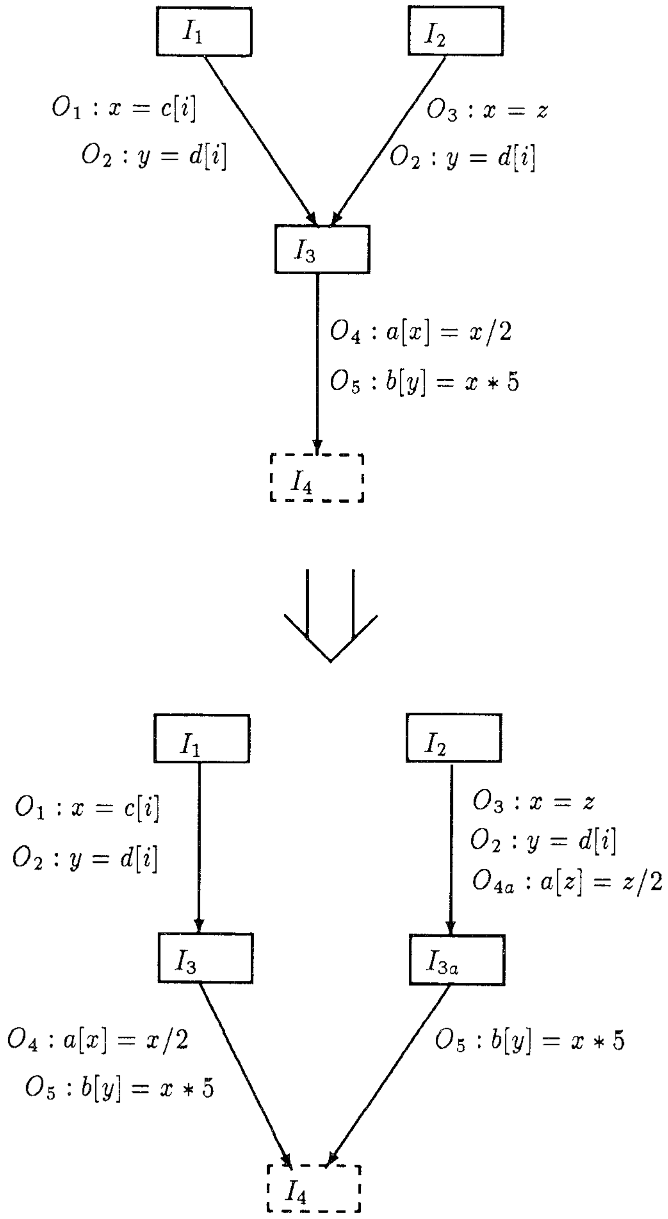


Figure 47. Example of code duplication

designed to reduce code expansion. When Enhanced Pipeline Scheduling produces a pipeline assuming infinite resources, the instructions can become extremely large, as renaming and forward substitution are repeatedly removing movement blocking dependences. In order to reduce the work due to finding more parallelism than can be utilized, they recommend that the movement of operations be allowed only within a given number of instructions (termed the window size) from the current fence instruction. The window size can be set to a value that gives

Table 6. Comparison of Algorithms

	Lam	Path	Pred	EMS	Perfect	Petri	Vegdahl	EPS
Fractional Rates	no	no	no	no	yes	yes	yes	no
$dif > 1$	yes	yes	yes	yes	yes	yes	yes	no
$min > 1$	yes	yes	yes	yes	yes	dummy	dummy	dummy
Resource Constraints	yes	no	yes	yes	yes	yes	np	np
Complexity	poly	poly	poly	2^n	poly	poly	2^n	poly
Code Expansion	yes	no	no	yes	yes	yes	yes	yes
Limited Rate	yes	yes	yes	yes	no	yes	no	yes
Leading Chain Problem	no	no	no	no	yes	no	no	no
Regular	yes	yes	yes	yes	no	no	yes	no
Kernel Recognition	no	no	no	no	yes	yes	yes	no
Modulo	yes	yes	yes	yes	no	no	no	no

good results for a particular application. A disadvantage of this technique is that it can limit the number of iterations executing concurrently within $\mathcal{K}$. When the window size is small, new operations from future iterations are not within the movement window until several fence instructions have been filled, thus slowing down the rate at which they can migrate back up to the current fence instruction.

5. SUMMARY

Table 6 summarizes the attributes of the various algorithms. The column, Path, represents Zaky's path algebra algorithm. Pred represents the various predicated modulo scheduling algorithms. Perfect represents Aiken and Nicolau's Perfect Pipelining. EPS represents Enhanced Pipeline Scheduling.

The table is divided into sections. The upper section shows factors that influence efficiency of the target code produced or the practicality of the algorithm. The middle section shows factors that influence the code size or compile time. The bottom section lists attributes that are descriptive in nature and do not affect the efficiency of the code produced or the generality of the algorithm, but may help to classify the algorithms. This table is extremely useful not only in determining the best algorithms, but in suggesting possible modifications to enhance their capabilities.

The row labeled Fractional Rates indicates whether the algorithm achieves fractional initiation intervals without having to replicate before scheduling. All algorithms can achieve fractional initiation intervals with replication, but suffer from the inability to decide easily on an appropriate replication factor.

All algorithms either address *min* times of greater than 1 as an integral part of the technique or introduce dummy nodes to remove the occurrences. *Dif* values greater than 1 are handled by all algorithms except EPS which must replicate before scheduling to remove the occurrences. All algorithms deal with general resource constraints except for Zaky's path algorithm. This is a serious limitation of Zaky's algorithm. Resource constraints can be expressed by a limit on the number of functional units, a conflict graph, or a resource vector of heterogeneous resource requirements.

All methods handle irregular persistent resources except for Path (which ignores resource constraints) and those marked with *np*, which deal with nonpersistent resources only.

Most of the algorithms have complexity $O(n^3)$ or $O(n^4)$. However, Vegdahl's algorithm is $O(2^n)$ and is thus impractical for even moderate sized problems.

EMS has a worst case complexity of $O(2^n)$ resulting from the exponential code explosion, but such cases are rare.

Code expansion refers to the increase in code size caused by the same operation being repeated multiple times in the schedule (not counting prelude and postlude). Any method that achieves fractional rate does so by incurring code expansion. Lam's code expansion comes from scheduling operations that are not in branches with *both* versions of the branch code and from modulo variable expansion to eliminate loop-carried antidependences. The hardware support for predicated modulo scheduling eliminates this code expansion. The transformations EPS uses to reduce the length of cyclic dependences also introduces code expansion.

Limited Rate indicates whether the algorithm uses the idea of a target *II* in making decisions. All modulo methods begin with a lower bound on the *II* in order to save effort. In addition, because a single copy of each operation exists in the kernel, the rate is automatically limited by *II*. Kernel recognition algorithms often lack the ability to control the greediness in initiating iterations possibly resulting in greater compilation time, longer initiation intervals (greater execution time), and greater span (increased length of prelude and postlude). The Petri net algorithm achieves this limited rate via the pacemaker (but this is an enhancement to the algorithm rather than an integral feature). In Perfect, rate is indirectly limited by setting the maximum span of a single instruction as new operations cannot execute if the set span is exceeded. Vegdahl is exhaustive and searches for *all* schedules, but altering the algorithm to limit the rate of execution would greatly reduce the compile time. In EPS, a limited rate does not exist per se, but the rate is limited by the number of times an operation is moved past the back edge or the window size.

The Leading Chain Problem refers to the problem that some algorithms experience when there are one or more nodes that are not successors of operations in the critical cycle. A leading chain often creates problems in algorithms without a measured rate as they are unconstrained. Perfect solves the leading chain problem by delaying operations when it is evident they are executing at different rates.

It is possible for all methods to handle predicates within the loop when if-conversion is used. Only EPS is capable of achieving different *II* on each branch of a predicate. Lam schedules subproblems before combining. EPS and Perfect are transformation based. Petri and Path both can perform transformations prior to scheduling to improve pipelining results.

The row labeled Regular indicates whether an operation in the prelude or postlude is executed at the same interval as it is while executing the kernel. Inasmuch as Vegdahl's method does not specify how the prelude and postlude are generated, we assume a regular schedule is produced. A regular schedule can be both an advantage and a disadvantage. Because the prelude and postlude execute just like the kernel (only with some operations omitted), predicated execution can make kernel-only code possible. However, as some operations are omitted, separate scheduling of prelude and postlude may be more efficient. This is a minor point as the conversion between a regular schedule and an irregular schedule (and vice versa) is a trivial transformation.

Most of the algorithms are either modulo or kernel recognition, but EPS is neither.

5.1 Modulo Scheduling Algorithms

Rather than waiting for a pattern to form, modulo techniques analyze the DDG and create a desirable schedule. The tight coupling of scheduling and pipelining constraints results in a pipeline that is near optimal. The ability to adjust the scheduling technique to control register pressure or prioritize by various attributes gives great flexibility. The regular pipeline that is produced simplifies the formation of the prelude, kernel, and postlude. With the replication suggested, fractional rates can be achieved [Jones 1991]. The trial-and-error approach of finding the achievable *II* increases the compile time, but no software pipelining algorithm has a tight bound on compile time.

Zaky [1989] uses a technique that produces results very similar to other modulo scheduling techniques, but uses path algebra as a framework. Zaky's algorithm is impractical as it cannot handle resource constraints, but is important because of the elegant way it formulates and solves the problem. Because of the concise algorithm, this technique provides an excellent conceptual model.

The modulo scheduling techniques have continued to improve by using hardware support to reduce code expansion and allow tighter schedules by the use of rotating register files. These methods represent an excellent choice for software pipelining, their only drawback being that fractional rates are not achieved without replication *before* scheduling.

5.2 Perfect Pipelining

Perfect Pipelining is important from an historical perspective due to the early work in exploration of pipelining involving branches within the body of the loop. Unification is an important addition to earlier global code motion techniques.

Conceptually, Perfect Pipelining has an advantage in that it is not forced to consider all paths through the loop simultaneously. In other words, various paths through the loop are able to achieve a different initiation interval. The frequency of achieving optimal overall results is hampered by the ad hoc nature of the scheduling.

The main disadvantage of Perfect Pipelining is the difficulty of determining when two nodes are functionally equivalent. Other problems include the generation of loops that contain more copies of the original iteration than needed and the need to help the pattern develop.

5.3 Petri Net Model

The Petri net model is another excellent choice for software pipelining due to its strong mathematical orientation and flexibility in adapting to a wide variety of constraints. General reservation models pose no problem for this adaptable method. It produces excellent schedules, its only drawback being the need to search for a pattern.

5.4 Vegdahl

Vegdahl's method is an interesting theoretical tool. Because of its exponential complexity, it is not a practical technique, but serves as an "optimal" for small code size. The method could be adapted for use with persistent resources, but would significantly increase the run time. Perhaps Vegdahl's method could be used in combination with other techniques. For example, if modulo scheduling was used to determine span and *II*, Vegdahl's exhaustive technique could be greatly restricted to explore solutions with span and *II* slightly better than the achieved. A parallel implementation of a greatly reduced search space could make the algorithm reasonable for a broader class of problems.

5.5 Enhanced Pipeline Scheduling

Enhanced Pipeline Scheduling is unique. Not only does it deal with multiblock loops, but it always maintains a legal loop structure rather than trying to rebuild the loop. Enhanced Pipeline Scheduling handles general loops. Although renaming and forward substitution have been utilized for years, the degree to which they have been successfully employed in this technique is noteworthy.

Enhanced Pipeline Scheduling has several advantages over Perfect Pipelining. The algorithm increases the speed of convergence by retaining the loop construct and reducing the resulting code size. These techniques can also cause problems, as prematurely forcing operations to remain together can restrict the parallelism. These disadvantages are lessened if a majority of the loop dependences can be removed by the automatic renaming and combining. The most serious drawback is the unsuitability of the method for use with persistent resources. Because instructions are inserted or removed during the scheduling, persistent resources (that require a fixed set of resources a given offset from instruction initiation) cannot be accommodated. Fractional rates are not achieved.

6. CONCLUSIONS AND FUTURE WORK

Software pipelining is an effective technique for extracting parallelism from loops that thwart attempts to vectorize or divide the work across processors. Although the speedup is modest, it should be noted that software pipelining succeeds where other methods fail and can be applied after other techniques have extracted coarse grain parallelism. The variety of architectures benefiting from software pipelining underscores its importance.

Although an NP-complete scheduling problem, software pipelining has numerous effective heuristics that have been developed. Both resource conflicts and cyclic dependences produce a lower bound on the initiation interval. Such lower bounds can be computed in polynomial time. In considering algorithmic features such as low complexity, ability to deal with conditionals, accommodation of resource conflicts, and achievement of fractional initiation intervals, the current methods have various degrees of success.

Some researchers are applying artificial intelligence techniques to the problem of software pipelining. Beaty [1991] uses genetic algorithms for instruction scheduling. O'Neill [1994] and Allan and O'Neill [1994] use genetic algorithms and simulated annealing to solve the problem of software pipelining. In the future, we expect to see more improvements in the application of artificial intelligence techniques to scheduling problems.

ACKNOWLEDGMENTS

The authors would like to thank B. R. Rau for his insight into the history of modulo scheduling and his painstaking attention to the details of this paper. Thanks to the anonymous referees. The quality of the paper has been tremendously improved by their comments. This work was supported in part by the National Science Foundation grants CDA-9100788 and IRI-8901175

REFERENCES

AHO, A V, SETHI, R, AND ULLMAN, J D 1988 *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA

AIKEN, A. 1988 Compaction-based parallelization. Cornell University, Dept of Computer Science, Ph D. Thesis. Ithaca, NY

AIKEN, A. AND NICOLAU, A 1988a. A development environment for horizontal microcode *IEEE Trans. Softw. Eng. 14*, 5 (May), 584–594

AIKEN, A AND NICOLAU, A. 1988b Optimal loop parallelization In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation* (Atlanta, GA, June), 308–317

AIKEN, A. AND NICOLAU, A. 1988c. Perfect pipelining: A new loop optimization technique. In *Proceedings of the 1988 European Symposium on Programming, Springer Verlag Lecture Notes in Computer Science, #300* (Atlanta, GA, March), 221–235

AIKEN, A. AND NICOLAU, A. 1990 A realistic resource-constrained software pipelining algorithm Tech. Rep. RJ 7743, IBM Research Division, San Jose, CA, October

ALLAN, V. H. AND O'NEILL, M. R. 1994. Software pipelining: A genetic algorithm approach. In *PACT 94* (Montreal, Canada, Aug. 23–26). North-Holland, Amsterdam, 311–314.

ALLAN, V. H, RAJAGOPALAN, M, AND LEE, R M. 1993. Software pipelining: Petri net pacemaker. In *Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism* (Orlando, FL, Jan 20–22). North-Holland, Amsterdam, 15–26

BANERJEE, U., SHEN, S., KUCK, D J., AND TOWLE, R. A 1979 Time and parallel processor bounds for Fortran-like loops. *IEEE Trans Comput C-28*, 9 (Sept), 660–670

BEATY, S. J. 1991. Instruction scheduling using genetic algorithms. Colorado State University, Fort Collins, CO, Ph.D Thesis.

BECK, G. R., YEN, D W L., AND ANDERSON, T L 1993 The Cydra 5 minicomputer: Architecture and implementation. In *J. Supercomput* (May), 143–180

BRETERNITZ, M., JR. 1991. Architecture synthesis of high-performance application-specific processors. Carnegie Mellon University, Pittsburgh, PA, Ph D Thesis.

CHANDY, K. M. AND KESSELMAN, C. 1991. Parallel programming in 2001. *IEEE Softw. 8*, 6 (Nov.), 11–20.

DAVIDSON, E. S. 1971. The design and control of pipelined function generators. In *Proceedings of the 1971 International IEEE Conference on Systems, Networks, and Computers* (Oaxtepec, Mexico, Jan.), 19–21.

DEHNERT, J. C., HSU, P. Y.-T., AND BRATT, J. P. 1989. Overlapped loop support in the Cydra-5. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, April), 26–38.

DEHNERT, J. C. AND TOWLE, R. A. 1993. Compiling for the Cydra-5. In *J. Supercomput.* (May), 181–228.

EBCIOĞLU, K. 1987. A compilation technique for software pipelining of loops with conditional jumps. In *Proceedings of the 20th Microprogramming Workshop (MICRO-20)* (Colorado Springs, CO, Dec.), 69–79.

EBCIOĞLU, K. AND NAKATANI, T. 1990. A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture. In *Languages and Compilers for Parallel Computing*, D. Gelernter, Ed., MIT Press, Cambridge, MA, 213–229.

FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (July), 319–349.

FISHER, J. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput. C-30*, 7 (July), 478–490.

FREUDENBERGER, S. M., GROSS, T. R., AND LOWNEY, P. G. 1994. Avoidance and suppression of compensation code in a trace scheduler *ACM Trans. Program. Lang. Syst. 16*, 4 (July), 1156–1214.

GAO, G. R., WONG, W.-B., AND NING, Q. 1991a. A timed Petri-net model for fine-grain loop scheduling. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* (June 26–28), 204–218.

GAO, G. R., WONG, W.-B., AND NING, Q. 1991b. A timed Petri-net model for fine-grain loop scheduling. Tech. Rep. ACAPS Technical Memo 18, School of Computer Science, McGill University, Montreal, Canada, H3A 2A7, January.

GOVINDARAJAN, R., ALTMAN, E. R., AND GAO, G. R. 1994. Minimizing register requirements under resource constrained rate-optimal software pipelining. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, to appear.

HSU, P. Y. T. 1986. Highly concurrent scalar processing. University of Illinois, Urbana-Champaign, Urbana-Champaign, IL, Ph.D. Thesis.

HSU, P. Y. T. AND DAVIDSON, E. S. 1986. Highly concurrent scalar processing. In *Proceedings of the Thirteenth Annual International Symposium on Computer Architecture*, 386–395.

HUFF, R. A. 1993. Lifetime-sensitive modulo scheduling. In *Conference Record of SIGPLAN Programming Language and Design Implementation* (Albuquerque, N.M., June). ACM, New York, 258–267.

JONES, R. B. 1991. Constrained software pipelining. Utah State University, Dept. of Computer Science, Logan, UT, Master's Thesis, Sept.

JONES, R. B. AND ALLAN, V. H. 1990. Software pipelining: A comparison and improvement. In *Proceedings of the 23rd International Symposium and Workshop on Microprogramming and Microarchitecture (MICRO-23)* (Orlando, FL, Nov. 27–29). IEEE Computer Society Press, 46–56.

KUCK, D. J. AND PADUA, D. A. 1979. High-speed multiprocessors and their compilers. In *Proceedings of the 1979 International Conference on Parallel Processing*, 5–16.

LAM, M. S. 1988. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation* (Atlanta, GA, June), 318–328.

LAM, M. S. 1987. A systolic array optimizing compiler. Carnegie Mellon University, Dept. of Computer Science, Pittsburgh, PA, Ph.D. Thesis.

MAHLKE, S. A., LIN, D. C., CHEN, W. Y., HANK, R. E., AND BRINGMANN, R. A. 1992. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25)* (Portland, OR, Dec. 1–4), 45–54.

MATETI, P. AND DEO, N. 1976. On algorithms for enumerating all circuits of a graph. *SIAM J. Comput. 5*, 1, 990–999.

NAKATANI, T. AND EBCIOĞLU, K. 1990. Using a lookahead window in compaction-based parallelizing compiler. In *Proceedings of the 23rd Microprogramming Workshop (MICRO-23)* (Orlando, FL, Nov.). IEEE Computer Society Press, 57–68.

NAKATANI, T. AND EBCIOĞLU, K. 1989. "Combining" as a compilation technique for VLIW architectures. In *Proceedings of the 22nd Microprogramming Workshop (MICRO-22)* (Dublin, Ireland, Aug.), ACM, New York, 43–55.

NICOLAU, A. AND POTASMAN, R. 1990. An environment for the development of microcode for pipelined architectures. In *Proceedings of the 23rd Symposium and Workshop on Microprogramming and Microarchitecture* (Orlando, FL, Nov.), 69–79.

O'NEILL, M. R. 1994. Software pipelining with stochastic search algorithms. Utah State University, Dept. of Computer Science, Logan, UT, Master's Thesis.

RAJAGOPALAN, M. AND ALLAN, V. H. 1994. Specification of software pipelining using Petri nets *Int. J. Parallel Process 3*, 22. 279–307.

RAU, B. R 1994 Iterative modulo scheduling: An algorithm for software pipelined loops In *Proceedings of Micro-27, The 27th Annual International Symposium on Microarchitecture* (San Jose, CA, Nov 29–Dec. 2). ACM, New York, 63–74

RAU, B. R. AND FISHER, J. A. 1993. Instruction-level parallel processing: History, overview, and perspective. *J. Supercomputing 7*, 9–50.

RAU, B. R., LEE, M, TIRUMALAI, P. P., AND SCHLANSKER, M. S 1992. Register allocation for modulo scheduled loops: Strategies, algorithms, and heuristics. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation* (San Francisco, CA, June). ACM, New York, 283–299.

RAU, B. R., SCHLANSKER, M S, AND TIRUMALAI, P. P 1992. Code generation schema for modulo scheduled loops. In *Proceedings of Micro-25, The 25th Annual International Symposium on Microarchitecture* (Dec.). IEEE Computer Society Press, 158–169

RAU, B. R., YEN, D. W L., YEN, W., AND TOWLE, R. A. 1989. The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs *IEEE Comput.* (Jan.), 12–25.

RAU, B. R. AND GLAESER, C. D. 1981 Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing In *Proceedings of the Fourteenth Annual Workshop of Microprogramming* (Oct.), 183–198

RAU, B. R., GLAESER, C. D., AND GREENWALT, E. M. 1982. Architectural support for the efficient generation of code for horizontal architectures. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems* (March), 96–99.

SMITH, H. F. 1987. *Data Structures Form and Function* Harcourt Brace Jovanovich, San Diego, CA

SU, B., DING, S., WANG, J., AND XIA, J. 1987. GURPR—A method for global software pipelining. In *Proceedings of the 20th Microprogramming Workshop (MICRO-20)* (Colorado Springs, CO, Dec), 97–105.

SU, B, DING, S., AND XIA, J. 1986. URPR—An extension of URCR for software pipelining. In *Proceedings of the 19th Microprogramming Workshop (MICRO-19)* (New York, Oct.), 104–108.

TARJAN, R. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput. 1*, 2 (June), 146–160

TIERNAN, J. C. 1970. An efficient search algorithm to find the elementary circuits of a graph. *Commun. ACM 13*, 1, 722–726.

TIRUMALAI, P., LEE, M., AND SCHLANSKER, M. S. 1990. Parallelization of loops with exits on pipelined architectures. In *Proceedings of SuperComputing '90* (Amsterdam, Nov.), ACM, New York, 200–212

TOKORO, M, TAMURA, E., TAKASE, K., AND TAMARU, K. 1977. An approach to microprogram optimization considering resource occupancy and instruction formats. In *Proceedings of the 10th Annual Workshop on Microprogramming* (Niagara Falls, NY, Nov.), 92–108.

VEGDAHL, S. R. 1992. A dynamic-programming technique for compacting loops. In *Proceedings the 25th Annual International Symposium on Microarchitecture (MICRO-25)* (Portland, OR, Dec). IEEE Computer Society Press, 180–188

VEGDAHL, S. R. 1982. Local code generation and compaction in optimizing microcode compilers. Carnegie-Mellon University, Dept. of Computer Science, Pittsburgh, PA, Ph.D. Thesis

WARTER, N. J., HAAB, G. E., AND BOCKHAUS, J. W. 1992 Enhanced modulo scheduling for loops with conditional branches In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25)* (Portland, OR, Dec. 1–4), IEEE Computer Society Press, 170–179

WARTER, N. J., MAHLKE, S. A, HWU, W. M W., AND RAU, B. R 1993. Reverse if-conversion. In *PLDI* (Albuquerque, NM, June). ACM, New York, 290–299

WOLFE, M. J. 1990. *Tiny—A Loop Restructuring Tool, User Manual* Oregon Graduate Institute of Science and Technology, Beaverton, OR.

WOLFE, M J 1989 *Optimizing Supercompilers for Supercomputers* MIT Press, Cambridge, MA.

WOOD, W. G. 1979. Global optimization of microprograms through modular control constructs. In *Proceedings of the 12th Microprogramming Workshop (MICRO-12)* (Hershey, PA, Nov.), 1–6.

ZAKY, A. M. 1989. Efficient static scheduling of loops on synchronous multiprocessors. Ohio State University, Dept. of Computer and Information Science, Columbus, OH, Ph.D. Thesis.

ZIMA, H. AND CHAPMAN, B. 1991. *Supercompilers for Parallel and Vector Computers* ACM, New York.

Received November 1993, revised version received February 1995, final revision accepted July 1995.