Fine-grain Compilation for Pipelined Machines

Keshav Pingali*

TR 88-934
August 1988

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

* Supported in part by NSF grants DCR-8502884, CCR-8704367, ONR grant N00014-86-K-0215 and the Cornell NSF Supercomputing Center. Also supported by NSF grant CCR-8702668 and an IBM Faculty Development Award.

Fine-grain compilation for pipelined machines

Alexandru Nicolau[†]

Keshav Pingali[‡]

Alexander Aiken

Computer Science Department

Cornell University

Ithaca, NY 14853

Abstract

Computer architecture design requires careful attention to the balance between the complexity of code scheduling problems and the cost and feasibility of building a machine. In this paper, we show that recently developed software pipelining algorithms produce optimal or near-optimal code for a large class of loops when the target architecture is a *clean pipelined* parallel machine. The important feature of these machines is the absence of structural hazards. We argue that the robustness of the scheduling algorithms and relatively simple hardware make these machines realistic and cost-effective. To illustrate the delicate balance between architecture and scheduling complexity, we show that scheduling with structural hazards is NP-hard, and that there are machines with simple structural hazards for which vectorization and the software pipelining techniques generate poor code.

Keywords: Pipelined machines, instruction scheduling, software pipelining, compilation, scheduling complexity.

To appear in the Journal of Supercomputing

[0][†]Supported in part by NSF grants DCR-8502884, CCR-8704367, ONR grant N00014-86-K-0215 and the Cornell NSF Supercomputing Center.

[‡]Supported by NSF grant CCR-8702668 and an IBM faculty Development Award.

1 Introduction

The generation of high-quality code for sequential and parallel machines is a challenging task. While much progress has been made towards this goal, generating optimal or provably close to optimal code for these machines has proven elusive. This is largely due to the intrinsic complexity of the problems involved, many of which are NP-Complete[Garey and Johnson 1979]. The inability to generate optimal code for all but the most idealized—and largely unrealistic—machine models has, in turn, negatively influenced the hardware design of machines: idiosyncratic designs and structural hazards are all too common. These, in turn, have compounded the difficulty of code generation. In principle, hardware hazards can be virtually eliminated, but there has been no real motivation to incur the extra expense since no code scheduling technique could produce provably optimal results for a reasonably general class of programs.

In this paper we show that the problem of generating optimal code for a pipelined machine with structural hazards is NP-Complete. We then investigate the applicability of a recently proposed loop scheduling technique [Aiken and Nicolau 1988a,b] to the problem of pipeline scheduling. The original technique produces idealized *parallel schedules*[1] which run in optimal time on a machine with enough processors to accommodate the schedule. The results we present in this paper investigate the applicability and limitations of these idealized schedules, when mapped to a machine with a single pipeline. We prove that for a large class of loops, the idealized schedules can be adapted to produce optimal code for hazard-free single or multiple processors. Furthermore, we show that even in cases where optimal machine schedules are not achievable, the optimality of the idealized schedule leads to a tight bound on the overall performance and good empirical performance. If structural hazards are introduced, we show that the schedule may be suboptimal even when an optimal schedule exists for hazard-free machines. However, we show that in such cases, an optimal fixed-size schedule for the loop—i.e., one whose size is not a function of the number of iterations in the loop—is not obtainable in general.

Finally, we present some empirical results based on the Livermore Loops indicative of the

[1]A parallel schedule is a static specification of a correctness-preserving partial order on operations. Each level in the schedule corresponds to a set of operations that can execute simultaneously.

empirical performance of our techniques for hazard free pipelined machines. These results are directly relevant to VLIW's [Fisher and O'Donnell 1984], pipelined machines, microcode engines, and in general to synchronous and synchronization-masking machines.

2 Optimal Scheduling of Operations for a Single Pipeline

The problem of scheduling operations for pipelined machines is closely related to classical problems in job-shop scheduling. For most real machines, there is at least one shared resource that cannot be accessed simultaneously by more than one operation —for example, memory (if there is no interleaving) or memory banks (if there is interleaving). We show that generating optimal code for a pipelined machine with such resource conflicts is NP-complete. The input to the problem is a *constraint graph*: a graph where nodes represent operations and edges represent dependencies between operations. Edges are labeled with integers, representing operation latencies. Directed edges represent data dependencies: if (u, v) is a directed edge labeled by integer i, then operation u must be scheduled before operation v and at least i operations (possibly noops) must be scheduled between u and v. Undirected edges represent resource constraints; if (u, v) is an undirected edge labeled by the integer i, then operations u and v can be scheduled in any order but must be separated by at least i operations. If u and v are nodes and (u, v) is an edge labeled i, we say that the *length of the interlock* between u and v due to edge (u, v) is i.

Definition: The *code reorganization problem* for a single pipeline is the following:

Input: a constraint graph.

Output: a sequence of operations i_1, i_2,..., with a minimum number of noops such that

1. if noops are deleted from the sequence, the result is a topological sort of the graph obtained by deleting undirected edges from the constraint graph
2. any two operations are separated at least by the length of the interlock between them

Theorem 1: The code reorganization problem is NP-complete.

Proof: We show that the following NP-complete resource constrained job-shop problem is reducible to the code reorganization problem. Given:

1. two processors
2. a set of jobs each taking one time unit for completion
3. a resource R that can be accessed by at most one job at each time step
4. an arbitrary DAG representing the precedence constraints between jobs
5. a deadline D.

Question: Is there a legal schedule for the jobs that meets the deadline?

This problem is known to be NP-complete [Ullman 1975]. To solve this problem, we can construct a code reorganization problem by adding the following nodes and edges to the precedence graph:

1. label all edges in the precedence graph by "1".
2. if u and v are jobs that require the resource, add an undirected edge (u, v) labeled "1"
3. add a chain of D nodes, called the ***serializer chain***, in which edges are labeled "2".
4. add a start node s and edges labeled "0" from the start node to each node in the precedence graph without any ancestors, and an edge labeled "2" from the start node to the first node in the serializer chain.

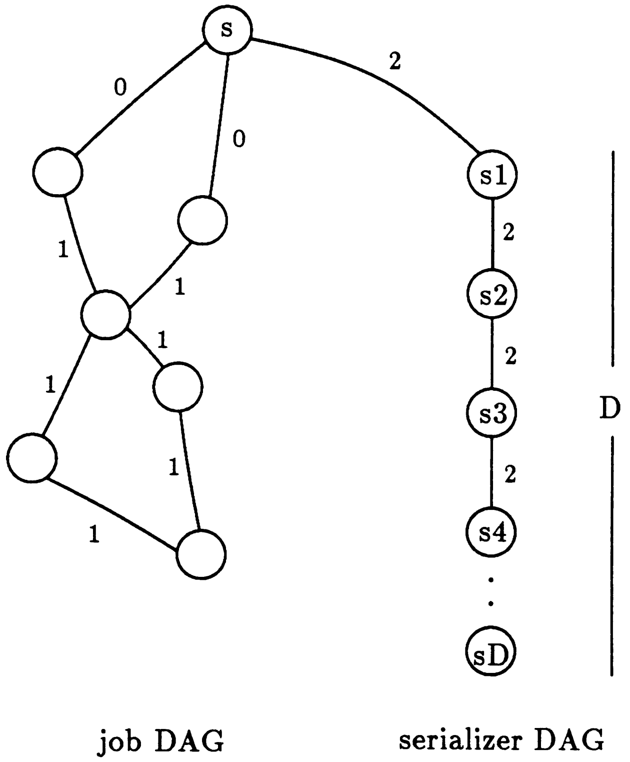
Figure 1 shows an example of this construction.

If there is a schedule for the job-shop problem that meets the deadline, then the result of the code reorganization must look like the schedule shown in Figure 1(b), where n_1, n_2, etc. represent nodes from the job-shop problem or noops. The solution for the job-shop problem can then be constructed as follows

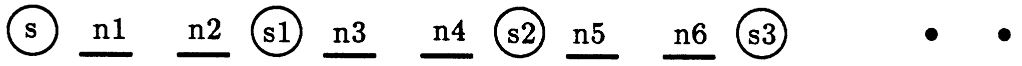
processor	step1	step2...
p1	n_1	n_3 ....
p2	n_2	n_4 ....

This must be a legal schedule for the job-shop problem since

1. precedence constraints between jobs are respected; if (u, v) is a directed edge, note that u and v cannot be scheduled in the same time step.
2. two jobs that use the resource cannot be scheduled in the same time step; if (u, v) is an undirected edge, u and v cannot be scheduled in the same time step.



(a) Construction of Code Reorganization Problem from Job-shop Problem



(b) Optimal Schedule for Code Reorganization Problem

Figure 1: Constructing a Code-reorganization Problem from the Job-shop problem

Therefore, the specified resource constrained job-shop problem is reducible to the code reorganization problem. If the optimal solution to the code reorganization problem has the form shown, the schedule for the job-shop problem can be read off; otherwise, there is no schedule for the job-shop problem that meets the given deadline. Therefore, the code reorganization problem is NP-complete. □

This result would not have been disappointing until recently, since software techniques for code scheduling did not usually produce provably optimal code even within idealized models [Fisher etal 1984], and even when they did, it was these results were not directly relevant at the machine instruction level. Recently however, techniques have emerged that produce absolute time-optimal parallel schedules for loop execution on clean multiple pipeline machines subject to the data-dependencies of the loop and the availability of enough resources (i.e., pipelines) to accommodate the schedule.

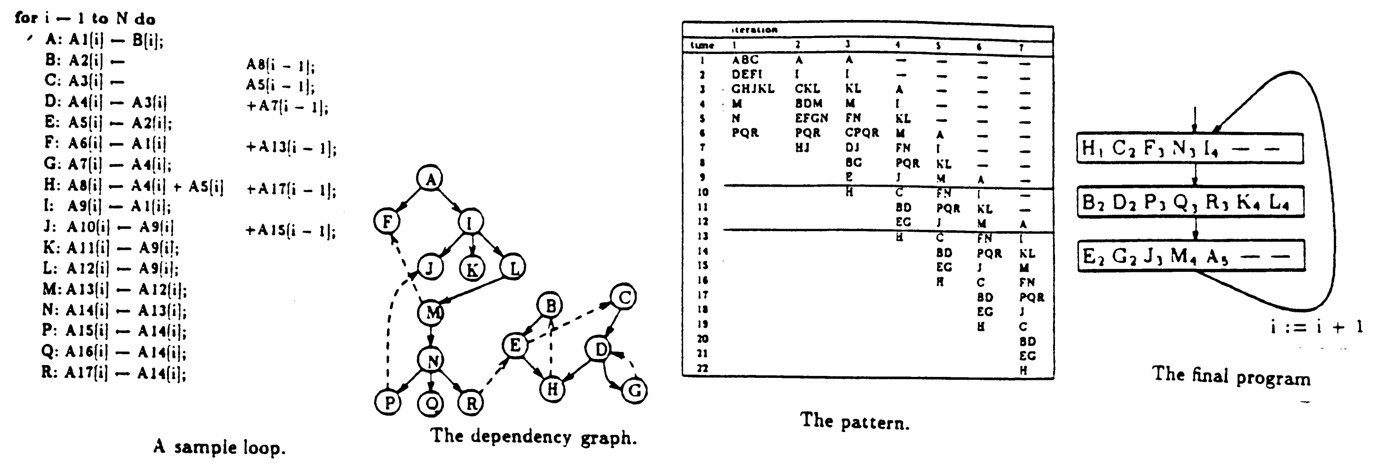
The most general technique, Perfect Pipelining, combines the benefits of fine-grain parallelism (can exploit irregular forms of parallelism) with the pipelining of iterations of coarser methods [Cytron 1986]. Perfect Pipelining uses incremental unwinding and successive applications of parallelization (compaction) transformations (e.g., Percolation Scheduling [Nicolau 1984b]), to detect a *pattern* in the code—which in practice emerges after a small amount of unwinding. The loop body can then be replaced by this pattern yielding a *schedule* for the loop. It can be shown that given enough resources, and subject to the given compaction transformations, the resulting loop will yield the best (optimal) running time, (i.e., further unwinding and compaction of the loop cannot yield better speedups). In particular, the running time of the new loop is identical to what might be obtained by full unwinding of the loop and full fine-grain parallelization, if such unwinding is feasible. This is important, since in practice loops can (or should) seldom be fully unwound at compile-time. These results hold even in the presence of conditional jumps and multicycle operations.

The second technique, Optimal Loop Parallelization (OPT), deals with loops containing no conditionals, or in which conditionals are removed [Allen and Kennedy 1983] (or the probability of paths execution is predictable). For such loops, OPT, which is a refinement of perfect pipelining, achieves an even stronger result. We can show that given *any* parallelization transformations that preserve the original data-dependencies, our transformation achieves equal or better running time for the final loop. In other words, OPT not only yields the best running time for the loop with respect to unwinding and the particular parallelizing transformations used, but true time optimality with respect to any possible dependency-preserving transformations[2]. OPT relies on the fact that only a small number of iterations ever need to be examined to determine a pattern which yields an optimal running-time schedule for the loop. These results hold in the presence of multicycle operations. The justification of these claims and the details of the algorithm are given in[Aiken and Nicolau 1988b]. For the purpose of this paper we only need to understand how OPT works. OPT incrementally unwinds the loop, allowing operations to be scheduled as early as possible in the schedule, subject only to data-dependencies and latencies[3]. Thus operations are scheduled at the earliest possible time they could be issued at runtime, if a synchronous multiprocessor were available. In [Aiken and Nicolau 1988b], it is shown that a repeating, fixed size pattern is guaranteed to emerge after a small amount of such unwinding and compaction, if the original data-dependencies of the loop are not allowed to drastically change throughout the process. Further unwinding and compaction beyond this point cannot improve parallelism, and thus replacing the loop body with this pattern will yield an optimal execution schedule for the given loop. Of course, a prologue and an epilogue including some start-up and wind-down code may be required; this code is extracted automatically as part of the algorithm. Other details such as the loop overhead can be handled with either hardware or software. The software approach has been discussed in detail in [Aiken and Nicolau 1988b], while some hardware mechanisms which would be relevant have been implemented and are discussed in [Cydrome 1987; Ebcioglu 1987].

In the remainder of the paper, we concentrate on OPT, since it produces absolute time-optimality for the loops it applies to. The arguments apply equally to PP, but then we need to take into account that the optimality is only with respect to the particular transformations

[2]Dependency changes (e.g., due to renaming) can be allowed in this context, even if done dynamically as part of the parallelization process.

[3]We are essentially performing a topological sort, creating a partial ordering of the operations; operations that are placed at the same level in the schedule are therefore independent of each other and can be executed in parallel. Given a synchronous parallel processor with enough resources, such a schedule could run "as is", with each level or slice of the schedule issuing each cycle



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

A sample loop.

The dependency graph.

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

The pattern.

$H_1\ C_2\ F_3\ N_3\ I_4$ — —
$B_2\ D_2\ P_3\ Q_3\ R_3\ K_4\ L_4$
$E_2\ G_2\ J_3\ M_4\ A_5$ — —
i := i + 1

The final program

Figure 2: Sample OPT scheduling

used. An illustration of the effects of OPT and the optimal schedule produced for the given loop is found in Figure 2. This example is taken from [Cytron 1984], where a heuristic schedule is developed for the same loop. For simplicity, and in keeping with the original example, latencies of operations in this example are assumed to be one cycle. As mentioned earlier, OPT can deal with realistic operation latencies.

When taking into account true operation latencies the notion of optimality derived from OPT/PP is realistic, in the sense that a schedule produced by OPT or PP could be run "as is" on a synchronous parallel machine (e.g., Trace-28 [Multiflow 1987]), with enough resources. Still, in practice, resources are often not available to allow the direct execution of the optimal schedule, and thus a *mapping* phase that adapts the optimal schedule to the actual hardware is needed. The more idiosyncratic the hardware (e.g., structural hazards, non-uniform pipes), the harder this mapping becomes. In this context, it is natural to ask what the relevance of the optimal schedules is for practically feasible machines. In this light, the result of Theorem 1 is discouraging.

However, this does not mean that the idealized schedules are not useful. A given optimal schedule could be run "as is" on machines with enough parallelism to support it. In particular, loops that yield only small degrees of parallelism—and thus have optimal schedules requiring relatively few resources—can be accommodated directly on even small parallel/pipelined engines, with a guarantee of optimal running-time. In fact, optimality is most important for such loops, since the parallelism they yield is small to begin with, and any degradation in performance—due to poor heuristic scheduling—is proportionately more significant. Even for loops with optimal schedules requiring more resources than the actual machine can provide, the (idealized) OPT schedule is a good starting point for a heuristic mapping, and serves as a yardstick by which to measure the final heuristic schedule. Furthermore, Theorem 1 does not preclude optimal scheduling for machines with clean (i.e., hazard-free) hardware. In the remainder of the paper we explore these issues in the context of clean machines. Such machines are quite realistic, in the sense that the hardware for the basic CPU can be built at reasonable cost. Whether the hardware overhead involved is less or more than that typically tolerated in existing machines for other reasons (e.g., interlock detection), is an open question. While memory bank conflicts can also be seen as a hazard, they too can (through aggressive interleaving and latency masking techniques) be arbitrarily reduced.

3 Loops without Loop-Carried-Dependencies

An important class of loops is *recurrence-free* loops. Such loops—or loops that can be transformed into this form—are amenable to vectorization and account for a significant fraction of code encountered in practice— 50% – 70% according to some estimates [Hack 1986]. No *loop-carried dependencies (lcd's)* exist in such loops, each statement in the loop can be vectorized, or (equivalently) potentially all iterations of the loop could be executed in parallel, as in a *DOALL* loop [Cytron 1984]. Unfortunately, depending on the details of the architecture available, neither of these methods, by themselves, guarantees time-optimal execution.

Executing recurrence-free loops as vector statements requires special hardware, both to decode and support the execution of the vector statements, and to allow *chaining* of the vector operations. If resource constraints exist, the order in which the vector statements appear and thus the order in which they are issued at runtime may affect the overall running time of the loop. In fact, the optimal issue order may even require the issue of the statements in the loop body in an interleaved fashion, not corresponding to any ordering of the vector statements. On the other hand, vectorization often produces optimal code, assuming memory-to-memory operations, chaining, and a clean pipelined machine. A comparison between the techniques we propose and vectorization would thus have to include a careful analysis of the hardware support required by each of the techniques, and their range of applicability. Such a comparison is beyond the scope of this paper, which deals with the applicability of OPT idealized schedules to realistic machines.

Similarly, the *DOALL* mechanism doesn't necessarily yield an optimal schedule. If enough processors are available to execute all loop iterations (including any fine-grain parallelism inside the iterations) in parallel, then time optimality is trivially ensured. However, in any practical machine, the parallel execution of a limited number of iterations does not necessarily yield an overall optimal execution, given the resources at hand. The feasibility of scheduling the operations inside the iterations optimally is an open problem, believed to be NP-hard.

OPT has the same problem as the *DOALL* approach. However, we can circumvent the problem by utilizing some of the parallelism available in the loop to improve utilization for the realistic machine, rather than (superfluously) increasing parallelism in the idealized schedule. By doing this methodically, we can generate optimal code for a given realizable clean machine. The following theorems formalize this result. We assume that dependencies are fixed, determined by a disambiguation system [Banerjee 1979; Nicolau 1982a; Wolfe 1982]. The results can be extended to allow useful transformations such as incremental renaming, which may modify the dependency graph.

Lemma 1: Given a loop without lcd's, and restricting OPT scheduling so that

$$\forall op \in iteration_i, scheduledcycle(op) \geq i,$$

(i.e., iterations are staggered by one cycle), then the resulting schedule produced by OPT for the loop body is 1 cycle long, regardless of operation latencies and data-dependencies.

Proof: OPT normally schedules operations for earliest possible execution, subject to data-dependencies and operations latency. (Note that we are not yet considering limited resources.) In general, O_k, the set of operations scheduled at cycle k is given by:

$$O_k = \{op | k = Max(a, scheduledcycle(op')+latency(op'), ..., scheduledcycle(op'')+latency(op''))\}$$

where $op', ...op''$ are all the data-dependency predecessors of operation op. The value of a is the minimum schedule cycle for an operation in iteration i, and thus according to the lemma $a = i$ for $op \in iteration_i$.

For example, all operations in iteration one with no dependency predecessors are scheduled in cycle one, and these same operations from iteration two are scheduled in cycle two (rather than one), effectively delaying all operations in iteration two by one cycle with respect to those same operations in iteration one; in the absence of lcd's, no dependency interference between iterations exists, and thus nothing else can further influence the scheduling of the operations.

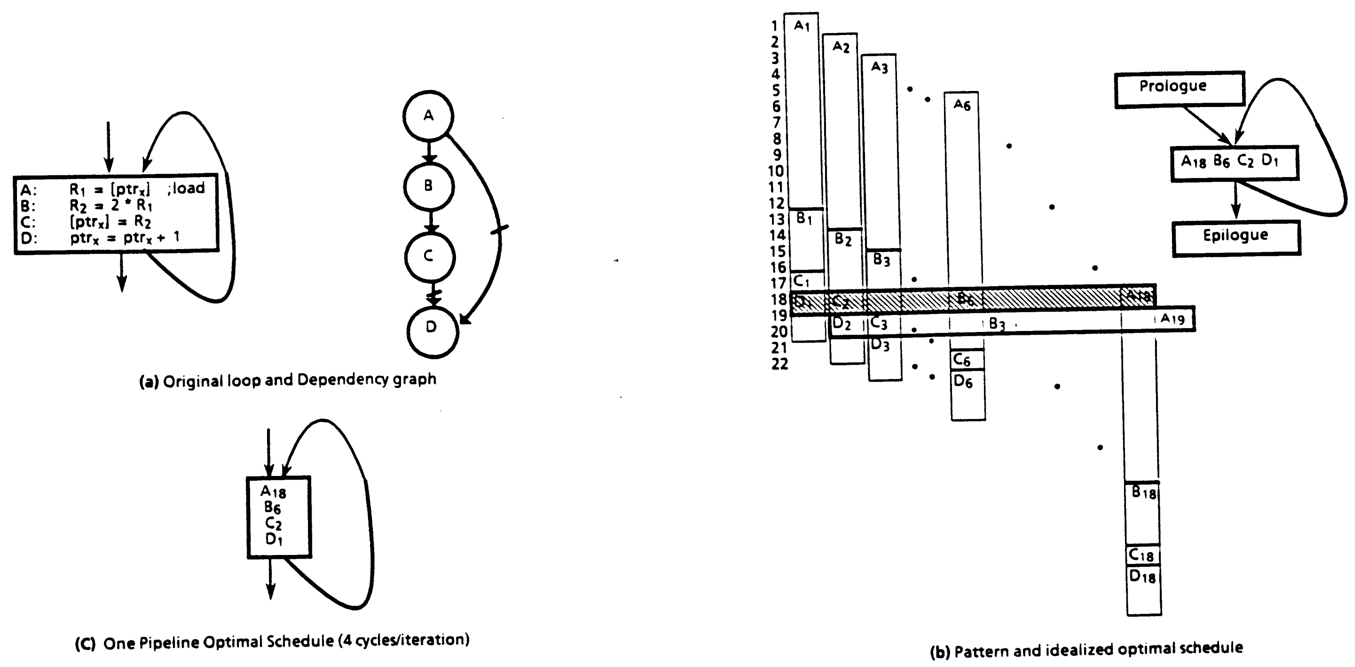
Assume that the last operation(s) in iteration one is scheduled in cycle j. Since j iterations are scheduled to start at one cycle intervals between cycle one and j, it follows that versions of all operations in the loop body (albeit from different iterations) are scheduled in cycle j. Thus cycle j contains the O_j set of iteration one, the O_{j-1} set of iteration two, and so on. After cycle j, the schedule doesn't gain further parallelism, since each iteration only contains j slices. Thus in cycle $j+1$, set O_j of iteration two, set O_{j-1} of iteration three, etc, are present, containing the same operations as the corresponding sets in cycle j . It is easy to see that as long as new iterations are available, this pattern repeats. Thus the pattern (schedule) by which OPT would replace the loop body is that derived from a single cycle j. $\square$

We note that there exists a direct linear-time algorithm for computing the OPT schedule for recurrence-free loops[Aiken and Nicolau 1988b]. An illustration is given in Figure 3a,b.

Theorem 2: Given a one cycle pattern as in Lemma 1, an optimal schedule for a 1-pipeline clean machine can be obtained efficiently.

Proof:

Te operations of the pattern can be issued (possibly with longest latency first to guarantee optimality including additive constants due to "wind-down/start-up" code encountered at the beginning and end of the loop) one per cycle, guaranteeing optimality. Note that this is possible since the operations in the pattern have no dependencies on one another, and that since latencies were already taken into account in deriving the schedule for the loop body, any operation from the next iteration can start immediately after any operation from the current iteration. Thus by linearizing the parallel loop body, (useful) operations are issued every cycle, and overall latency is minimized. This mapping yields optimal execution time of the loop for a one pipeline clean machine. $\square$



```
A:   R1 = [ptrx]   ;load
B:   R2 = 2 * R1
C:   [ptrx] = R2
D:   ptrx = ptrx + 1
```

(a) Original loop and Dependency graph

(C) One Pipeline Optimal Schedule (4 cycles/iteration)

(b) Pattern and idealized optimal schedule

Figure 3: Generating optimal schedules for loops w/o lcd's.

An example is shown in Figure 3c.

Corrolary: The algorithm in Theorem 2 can be extended to generate optimal schedules for a k-pipe clean machine (i.e., a machine that issue (any) k-operations per cycle).

Proof:

Let n be the number of operations in the pattern obtained by the modification of OPT in Lemma 1. If n is a multiple of k, then essentially the same approach as in Theorem 2 works: partition the operations in the pattern into groups of k, and let that be the loop body. Successive patterns need not be delayed due to this reordering, and thus the machine is kept running at full utilization and issuing the same operations as in the original code.

If n is not a multiple of k, find l, the smallest integer for which $n * l$ is a multiple of k. Since n is the number of operations in one iteration, modifying the restriction in Lemma 1 to:

$$\forall op \in iteration_i, scheduledcycle(op) \geq \lfloor i/(l+1) \rfloor + 1$$

creates a pattern with $n * l$ which, as we have seen, can be scheduled optimally. □

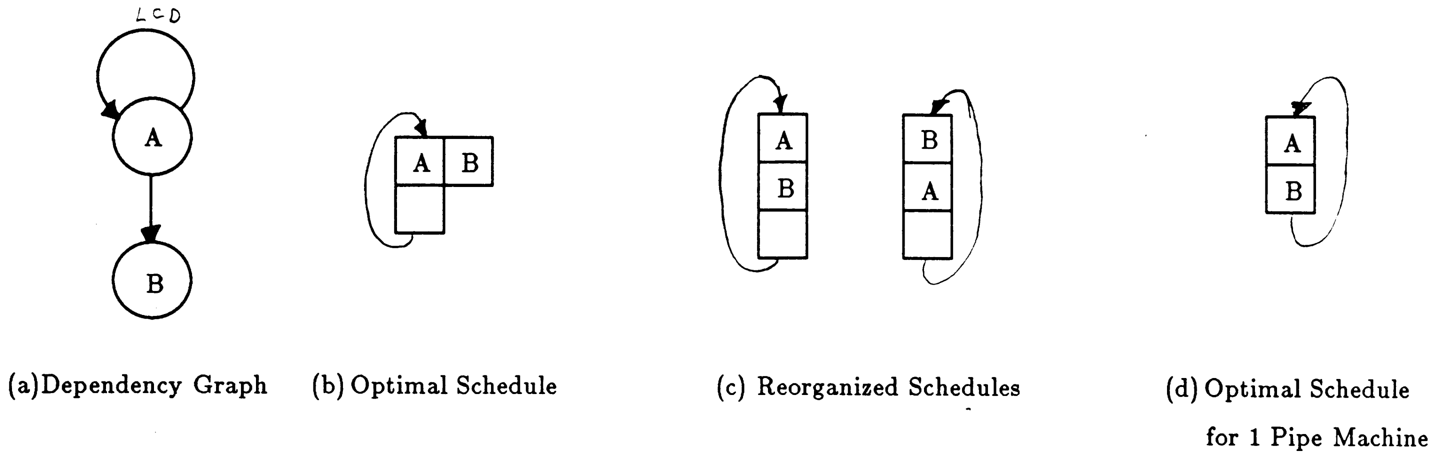
4 Loop-carried Dependencies

When loop-carried dependencies are present, there is less flexibility in manipulating the schedules obtained by OPT to achieve resource-constrained optimal schedules. In particular, the idealized optimal schedules may span multiple cycles, some of which may contain no operations. Empty cycles, or noops result from operation latencies, and thus cannot be removed from the schedule. If the target machine has enough parallelism to accommodate the schedule (i.e., if the largest number of operations that occur in any one cycle of the schedule is less than or equal to the number of operations that the processor can issue in parallel) then the schedule is still optimal for the given machine. Data-dependencies may allow some operations to occur in different places in the schedule without lengthening it and thus such operations may be rearranged to minimize resource requirements while preserving optimality. If this process suffices to bring the resource requirements within what is available in the given machine, we have a mapping to the actual hardware that guarantees optimal running time for the loop. If however the idealized schedule produced by OPT cannot, even with the above "tricks", be made to fit the resources available without lengthening the schedule, optimality cannot be guaranteed. This situation is illustrated in Figure 2. The initial schedule produced by OPT in the figure requires seven processors to execute "as is". However, a reasonable compiler could determine that operation Q has no dependents and thus can be delayed without lengthening the schedule. Note that this last schedule is optimal and fits on a 6-pipe clean machine. The various situations that may occur and their impact on the optimality of the realistic schedules produced are summarized in the following theorems. The term *transformed schedule* refers to any modifications we may perform on the optimal schedule—to limit resource requirements and/or to eliminate noops—without lengthening it.

Theorem 3 If the transformed schedule contains at least $n \geq 1$, or a multiple of n real operations (i.e., not noops) in every cycle, then an optimal realistic schedule is achievable for all k-pipe machines, such that $(n \mathit{\ mod\ } k) = 0$.

Proof: Immediate generalization of the mapping algorithm in the corollary to Theorem 2. $\square$.

Theorem 3 implies that when no noops are present, an optimal schedule is always feasible



(a)Dependency Graph (b) Optimal Schedule (c) Reorganized Schedules (d) Optimal Schedule for 1 Pipe Machine

Figure 4: Optimal mapping of OPT schedule doesn't preserve optimality

for a 1-pipe machine. Also note that when $n < k$, the idealized optimal schedule can execute as is, preserving optimality.

Mapping a schedule requiring k-pipes onto a machine with fewer pipes is a variation of a long-standing open problem, believed to be NP-Hard. Nevertheless, even if an efficient optimal mapping cannot be achieved, it could be argued that for very important—or small—inner loops, a brute force approach (e.g., integer programming) would be justified if such an optimal mapping of the schedule to the hardware would yield optimal running time for the loop. The following theorem dashes this hope by showing that even an optimal mapping of the schedule does not necessarily ensure overall optimal running time for the loop on a clean machine.

Theorem 4: Given a transformed schedule (containing noops) that requires k-pipes to guarantee optimal running-time for the loop, mapping the schedule to fewer than k pipes may not preserve optimality even if the mapping is optimal[4].

Proof: Figure 4 gives a simple example where an optimal mapping of the ideal schedule to resources does not yield optimal execution time. The OPT schedule is shown in figure 4a. No transformations of the schedule are possible without lengthening it. Two legal and

[4]For the mapping to be optimal it must lead to the smallest number of cycles for the mapped schedule.

minimal orderings (3 cycles per iteration) for the schedule are shown in Figure 4b. However, this does not yield an optimal running time for the loop. By considering two occurrences of the pattern, we can derive a schedule with only two cycles per (original) iteration, as shown in Figure 4c. Thus our conjecture that an optimal mapping of the optimal schedule yields optimal running time for the loop is false. □

So even if we start with an optimal schedule for k-pipes and go "all out" mapping it to fewer pipes, we lose optimality. Note that the trick we have used in the previous section to maintain optimality does not apply here: we cannot bring in independent iterations to fill the gaps. The schedule is optimal precisely because no further overlap (beyond what is already given by OPT) is possible between iterations. However, as we map to fewer resources, some operations are delayed while others are executed earlier, and the "stretched" schedule may no longer be optimal with respect to the overall loop, as operations may now be able to cross the boundaries between schedules, as illustrated in Figure 4.

While optimality cannot be preserved under these conditions, even with an optimal mapping, in practice even a simple heuristic mapping algorithm does very well. The following theorem shows that we can guarantee a bound of two times optimal using even the most naive mapping.

Consider a machine with k pipelined processors. A program for this machine consists of *instructions*—schedule cycles—which may contain up to k *operations*. An instruction is *full* if it has k operations.

Let L be a loop that is time-optimal, contains n instructions, and requires $k' > k$ processors. Let k_i be the number of operations in instruction i of L. For each instruction i, create $\lceil k_i/k \rceil$ instructions, where $\lfloor k_i/k \rfloor$ instructions are full and the last is (potentially) partially full. Let the resulting loop be L'.

Lemma 2: At most n instructions in L' are not full.

Proof: If an instruction in L is not full, then it contributes one partially full instruction to L'. Similarly, if an instruction in L has more than k operations, it contributes at most one partially full instruction to L'. So the total number of instructions in L' that are not full is at most n. □

Let $T_{L'}$ be the running time of loop L', and let T_L be the running time of loop L. Let

Loop	Original Code	Limited Processors	
	Mflops	1 proc Mflops	2 procs Mflops
LL1	9	31-50	57-100
LL2	8	20-35	40-60
LL3	7	16-20	20-23
LL4	6	16	20
LL5	6	12-15	15-16
LL6	8	6-16	6-20
LL7	20	36-51	71-99
LL8	11	40-55	80-110
LL9	17	35-49	68-97
LL10	10	18-25	36-48
LL11	4	4-9	4-11
LL12	4	13-20	27-40
LL13	4	11-12	22-24
LL14 (avg)	4	14-18	25-31
Average	8	19-28	35-50
Harmonic Mean	7	13-20	18-33

Table 1: Livermore Loops Results

T_* be the optimal running time of loop L rescheduled to run on a machine with k pipelined processors.

Theorem 5: $T_{L'}/T_* \leq 2$

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

5 Experimental Results

The factor of two bound in the previous section is based on a very naive mapping algorithm that makes no effort to eliminate noops in the schedule or to overlap successive patterns. overlap any successive schedules (iterations). A practical algorithm could certainly do more. Even with the naive scheduling algorithm however, a realistic clean machine does very well indeed, as illustrated by the results in Table 1.

We have implemented our scheduling algorithm, taking into account instruction latencies and processor limitations. Table 1 shows performance measurements for fourteen Livermore Loops [McMahon 1983]. The results are divided into two sections. Column two gives the flop rate of the original code on one pipelined processor, without statement reordering. It is included for reference. Columns three to four give the flop rate of the schedules computed by our algorithm for limited resources of one and two pipelined clean machines, respectively. The schedules were obtained by using the naive (worst-case) mapping algorithm described previously, and thus are lower bounds on the performance achievable with a reasonable mapping algorithm.

The speedups achieved may depend on the hardware and the extent to which the original code is optimized. For example, autoincrements issued as part of load instructions[5] and some sequential-code optimizations (such as removing redundant loads across iterations [Callahan etal 1987]) improve the speedups. To reflect this, some entries in Table 1 give a range. Standard compiler optimizations and the simplest addressing hardware achieve the lower number, while the optimization and hardware mentioned above achieve the higher number. The figure given for the original code is the better of the two approaches. For instance, in LL6, redundant load removal greatly improved the performance of the original code.

The flop rate for the original code is computed using the pipelining strategy of the Cray-1: instructions are issued in order as quickly as possible, subject to data dependencies. Even for a single processor, the improvement using our scheduling algorithm is dramatic. Half of the loops triple in performance when the additional optimization and hardware are assumed.

Interestingly enough, if a second pipeline is added and the in-order issue strategy of the Cray-1 is used, the original code runs only about 10% faster than that given in column two, so the relative speedup using our algorithm is much greater. Further increases in the number of pipelines available increases the performance obtained by our algorithm even further. The average number of processors required to directly execute the OPT schedules for these loops is twenty-two, and the average performance is $530 Mflops$.

These results are computed statically from the patterns generated by our algorithm. The effects of the preloop and postloop are not included, thus these results represent the asymptotic speedup for many iterations of the loop. Note, however, that the execution time is optimal for any number of loop iterations.

[5]This does not mean dedicated hardware for executing address increments. It simply means allowing load and increment operations to be combined into a single "macro-operation" issue, without changing the latencies of either of the two operations or their execution. By reducing the number of operations to be issued this approach improves the issue-rate, and thus the overall performance for some loops.

6 Structural Hazards

Theorem 1 virtually precludes the possibility of optimal code generation for machines with structural hazards. However, in the light of the results of the previous sections, a natural question to consider is whether the modifications to OPT can be extended to generate optimal code for lcd-free loops in the presence of structural hazards. Unfortunately, this is not feasible. Figure 5 presents a counterexample.

The loop body consists of two data-independent statements, one computing a multiply and the other an add. For simplicity —and to allow the computation to be somewhat meaningful—we assume that the statements perform memory-to-memory operations on vectors, the add taking two cycles to complete, the multiply three. Further, we assume a one-pipeline machine where the only structural hazard is that an add operation and a multiply cannot complete in the same cycle (i.e., an add cannot issue in the cycle following a multiply). Under these circumstances, OPT, as modified in Lemma 1, but not taking into account the structural hazards produces an idealized schedule in which add and multiply operations occur concurrently. This schedule maps optimally onto a clean 1-pipe machine, with add and multiply operations alternating. However, mapping this schedule to the target machine yields either of the schedules of Figure 5c, which leads to a noop being issued for every add-multiply pair. Of course, OPT can be modified to allow k iterations to fully overlap, yielding k add and k multiply operations. With the help of a reasonable mapping algorithm, only one noop needs to be generated, as shown in Figure 5d.) However, the optimal schedule for this loop is to issue all the add operations before all the multiplies, which is not easily derivable from the results of OPT. Of course, OPT *does* implicitly detect the fact that the operations are vectorizable, and thus could be easily modified to generate vector statements, which would achieve the desired result in this case. However, what is not clear is *when* to resort to this approach. For example, the hazard might have been that operations using the same operator cannot overlap (e.g., a new multiply may not start before the previous one has completed)—a natural requirement for a machine using multiple non-pipelined functional units. In this case, OPT generates optimal code, while vectorization does not.

The trend in architectural design is to avoid structural hazards as much as possible—a machine with too many structural bottlenecks cannot perform at or near its peak regardless

```
for i ← 1 to n
    A: A[i] ← A[i] + 1;
    B: B[i] ← B[i] * 2;
```

(a) Original program.

cycle	*operations*	
0	A_0	B_0
1	A_1	B_1
	...	...

(b) Idealized schedule.

cycle	*operation*
0	A_0
1	B_0
2	noop
3	A_1
4	B_1
5	noop
	...

(c) First legal schedule.

cycle	*operation*
0	B_0
1	noop
2	A_0
3	B_1
4	noop
5	A_1
	...

(d) Second legal schedule.

Figure 5: Structural Hazards

of the compiler technology used. We are arguing that the added (hardware) cost of avoiding structural hazards is justified by the existence of software techniques capable of generating optimal code for clean machines for large classes of loops, and provably good code for the cases where optimality is infeasible. If some structural hazards are present, then simple techniques such as further unwinding of the OPT schedule and compaction coupled with reasonable mapping algorithms can minimize the impact of the hazards on the quality of the code. This, coupled with the relative simplicity and uniformity of application of OPT/PP makes it a good candidate even for existing pipelined and synchronous parallel machines.

In practice, most structural hazards can be dealt with in software, by simply issuing operations in an appropriate order, interspersed with noops if necessary. Other hazards—such as memory bank conflicts—may require a combination of software/hardware mechanisms. In the first case, a simple bound can be derived for the mapping of our idealized schedules. The bound for lcd-free loops and hazards requiring one cycle delay between some operations is simply a factor of two over the clean—optimal—schedule, since at worst, all operations are scheduled one noop apart, guaranteeing no hazards are encountered. This bound can be trivially extended to hazards requiring more than one noop delay, with a corresponding degradation in performance. In practice, much better schedules are achieved by taking into account the hazards and heuristically mapping to minimize delays.

For cases where software techniques are not sufficient to handle hazards, (e.g., for memory contention) a fully static bound is unachievable directly. However, reasonable performance estimates can usually be made. For example, the average probability of a bank conflict can be estimated, and multiplied by the ensuing delay can yield the expected degradation resulting from such hazards. This metric can in turn be used to derive an expected bound on variation of the real machine's running time from a clean machine's running time. While such variations may not be completely eliminated in any real machine, a good design can make them arbitrarily small.

References

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

[2] Aiken, A. and Nicolau, A. 1988. Perfect Pipelining: A new loop parallelization technique. In *Proceedings of the 1988 European Symposium on Programming.* Springer Verlag Lecture Notes in Computer Science no. 300, March. Also available as Cornell Technical Report TR 87-873.

[3] Allen, J. R. and Kennedy, K. 1984. Automatic loop interchange. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, volume 19, pages 233–246, June.

[4] Allen, J. R., Kennedy K., Porterfield, C. and Warren, J. 1983. Conversion of control dependence to data dependence. In *Proceedings of the 1983 Symposium on Principles of Programming Languages*, pages 177–189, January.

[5] Banerjee, U. 1979. *Speedup of Ordinary Programs.* PhD thesis, University of Illinois at Urbana-Champaign, October. 79-989.

[6] Callahan, D., Cocke, J., and Kennedy, K. 1987. Estimating interlock and improving balance for pipelined architectures. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 297–304, August.

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

[8] Cytron, R. 1984. *Compile-time Scheduling and Optimization for Asynchronous Machines.* PhD thesis, University of Illinois at Urbana-Champaign.

[9] Cytron, R. 1986. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836–844, August.

[10] Ebcioğlu, K. 1987. A compilation technique for software pipelining of loops with conditional jumps. In *Proceedings of the 20th Annual Workshop on Microprogramming*, pages 69–79, December.

[11] Fisher, J. A., Ellis, J. R. Ruttenberg, J. C. and Nicolau, A. 1984. Parallel processing: A smart compiler and a dumb machine. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, pages 37–47, June.

[12] Fisher, J. A. and O'Donnell, J. J. 1984 VLIW machines: Multiprocessors we can actually program. In *Proceedings of CompCon Spring 84*, pages 299–305. IEEE Computer Society, February.

[13] Garey, M. R. and Johnson D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company.

[14] Hack, J. 1986. Peak vs. sustained performance in highly concurrent vector machines. *IEEE Computer*, September.

[15] McMahon, F. H. 1983. Lawrence Livermore National Laboratory FORTRAN kernels: MFLOPS. Livermore, CA.

[16] Multiflow Computer Inc., Branford, Connecticut. 1987. *Technical Summary*, 1987.

[17] Munshi, A. and Simons, B. 1987. Scheduling sequential loops on parallel processors. Technical Report 5546, IBM.

[18] Nicolau, A. 1984. *Parallelism, Memory Anti-Aliasing and Correctness for Trace Scheduling Compilers.* PhD thesis, Yale University.

[19] Nicolau, A. 1984. Percolation Scheduling: A parallel compilation technique. Technical Report 85-678, Cornell University.

[20] Wolfe, M. J. 1982. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, October.

[21] Ullman, J. 1975. NP-complete scheduling problems. *Journal of Computer and Systems Sciences* June.