Optimal Loop Parallelization*

Alexander Aiken
Alexandru Nicolau

Computer Science Department
Cornell University
Ithaca, New York 14853
aiken@svax.cs.cornell.edu

1 Introduction

Parallelizing compilers promise to exploit the parallelism available in a given program, particularly parallelism that is too low-level or irregular to be expressed by hand in an algorithm. However, existing parallelization techniques do not handle loops in a satisfactory manner. Fine-grain (instruction level) parallelization, or *compaction*, captures irregular parallelism inside a loop body but does not exploit parallelism across loop iterations.[1] Coarser methods, such as *doacross* [9], sacrifice irregular forms of parallelism in favor of pipelining iterations (*software pipelining*). Both of these approaches often yield suboptimal speedups even under the best conditions—when resources are plentiful and processors are synchronous.

In this paper we present a new technique bridging the gap between fine- and coarse-grain loop parallelization, allowing the exploitation of parallelism inside and across loop iterations. Furthermore, we show that, given a loop and a set of dependencies between its statements, the execution schedule obtained by our transformation is *time optimal*: no transformation of the loop based on the given data-dependencies can yield a shorter running time for that loop.

Our optimality results hold for synchronous parallel machines, such as horizontally microcoded engines, RISC architectures, the Mars-432, FPS-164/264, Multiflow's Trace series, Cydrome's Cydra, and Chopp. In addition, any multiprocessor supporting efficient synchronization and communication between processors (e.g., Alliant, Burton Smith's Horizon) will also benefit from our techniques. If the cost of synchronization and communication is low, then our method still produces code provably close to optimal.

The code generated by our algorithm makes efficient use of resources. In practice, a loop with n statements can be scheduled using less than n processors, depending on the parallelism available in the loop. If fewer resources are available than our algorithm requires for optimal speedup, optimality may no longer be achieved. (Scheduling with resource constraints is known to be NP-hard [12].) However, when compiling for very parallel machines (e.g., Multiflow's Trace-28, Smith's Horizon) the problem is often to find enough parallelism to effectively utilize the hardware. Even when resources are a limiting factor, an optimal schedule is a valuable guide in generating a legal schedule for the processors available.

*This work was supported in part by NSF grant CCR-8704367 and the Cornell NSF Supercomputing Center.

[1] While partial unwinding may alleviate this problem, it cannot eliminate it.

Proceedings of the SIGPLAN '88
Conference on Programming
Language Design and Implementation
Atlanta, Georgia, June 22–24, 1988

2 Motivation

Much attention has been devoted to the parallelization of *doacross* loops [9,16,17]. A doacross loop expresses some recurrence preventing the iterations of the loop from executing independently. In this paper we consider only doacross loops; a variation on our technique generates optimal code for loops with data independent iterations in the presence of resource constraints.

The most general problem can be summarized as: given a loop and its *dependency graph*, what is the best parallel schedule for the loop? The nodes of the dependency graph are the statements in the loop; the edges connect nodes that may access the same memory locations [13]. In general, the paradigm for approaching this problem has been to execute the iterations of a loop on separate processors, subject to the constraint that data dependencies between the loop iterations (*loop-carried dependencies* [4]) are not violated. A *delay*, d, is calculated to satisfy the following condition: if iteration $i+1$ is started d steps after iteration i, then all loop-carried dependencies are preserved.

A somewhat more general approach has also been studied: determining the optimal delay if dependency-preserving reorderings of the statements in the loop body are considered [16]. Computing the loop body with the optimal delay is NP-hard [8], so it would appear that a polynomial-time algorithm to compute optimal code is impossible. However, the intractability of the problem lies in the difficulty of finding the best ordering of statements in an iteration; this only shows that the general problem restricted to the case where iterations are scheduled as indivisible units is NP-hard.

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

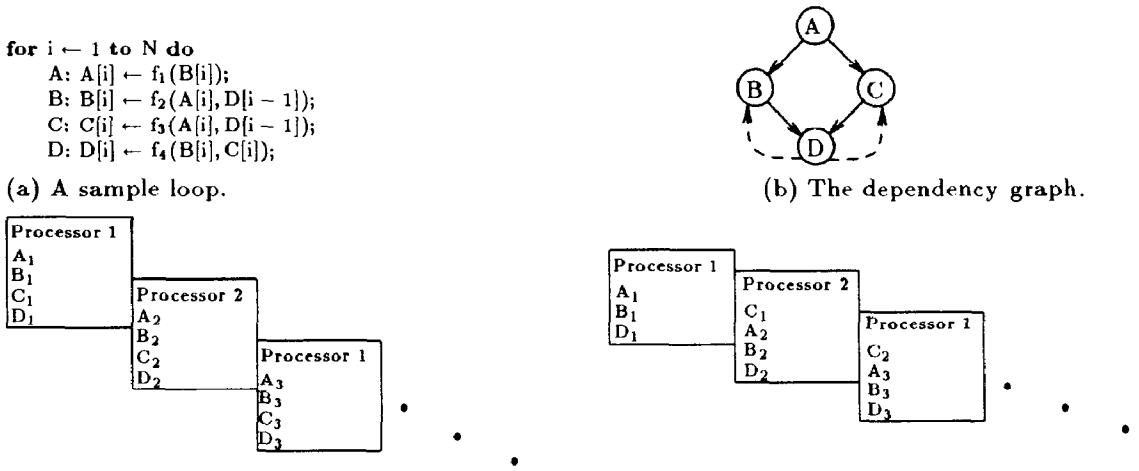
We make several standard assumptions about the loops to be scheduled. For simplicity, we assume that statements execute in a single machine cycle; there are straightforward extensions of our method for multi-cycle statements [2]. We assume that loop-carried dependencies are from one iteration to the next; Munshi and Simons have observed that loops encountered in practice can be converted to this form by unrolling [16]. An algorithm to perform this transformation automatically appears in [1]. Our algorithm applies to innermost loops; techniques for unrolling nested loops can be used to handle outer loops [1,7].

Our major restriction is that the loop body should contain no If-statements other than exit tests. There are techniques which essentially eliminate tests from a loop. *If-conversion* can be used to transform any loop into a semantically equivalent loop without tests [5]. Another method is to make assumptions about the branching behavior of tests and apply a transformation to the most important path(s) [6,11]. Recently, software pipelining techniques have been developed that handle tests directly [3,10,14]. It is an open problem whether the optimality results of this paper can be extended to loops with arbitrary flow of control.

3 The Approach

This work is based on results in compaction-based software pipelining [3]. To make this paper self-contained, we recast the general results of [3] in the special context of this paper. A program is represented by its dependency graph. For simplicity, we consider only *true dependencies* [13]; there is a true dependency between two statements if one statement writes a value that the other reads. Other types of dependencies (so-called *anti-* and *output* dependencies) present no additional problems.

The basic strategy behind our method is very simple. We examine a partial execution history of a loop, say the first i iterations, and schedule the statements of those i iterations as early as possible. We call this a *greedy schedule*. If the longest chain of dependencies on which a statement x depends has length j, then x is scheduled at time j. A loop's dependencies have



(a) A sample loop.

(b) The dependency graph.

(c) An optimal Doacross schedule.

(d) An optimal schedule.

Figure 1: An example.

some regularity, specified by the dependency graph. Thus, scheduling a large enough portion of a loop's execution history should reveal some repeating behavior, which (intuitively) can be used to obtain a good schedule for the loop. Informally, we refer to this repeating behavior as the *pattern* of the loop. We show that the portion of the execution history that must be examined to compute optimal schedules is small. This yields a polynomial time algorithm for computing optimal parallel schedules for a large class of loops and parallel machines.

4 Computing Patterns for Statements

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

Definition 4.1 A *dependency chain* is a sequence of statements $\boldsymbol{x}^1_{h_1}\boldsymbol{x}^2_{h_2}\ldots\boldsymbol{x}^k_{h_k}$ such that $(\boldsymbol{x}^i, \boldsymbol{x}^{i+1})$ is an edge in the dependency graph.

Definition 4.2 Let $C = \boldsymbol{x}^1_{h_1}\ldots\boldsymbol{x}^k_{h_k}$ be a dependency chain.

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

Definition 4.3 Let $C = \boldsymbol{x}^1\ldots\boldsymbol{x}^k\boldsymbol{x}^1$ be a cycle. Let p be the number of loop-carried dependencies—including the dependency $(\boldsymbol{x}^k, \boldsymbol{x}^1)$—in the cycle. The *slope* of C is the ratio k/p.

The slope of a cycle establishes a bound on the rate that statements in the cycle can be executed. Using C from Definition 4.3, statement $\boldsymbol{x}^i_{j+p}$ cannot be executed sooner than k steps after $\boldsymbol{x}^i_j$. (This notion of the slope of a cycle is due to Callahan, Cocke, and Kennedy [7].) In a greedy schedule, the two statements must be scheduled at least k steps apart. We write $dist(\boldsymbol{x}, y)$ for the number of steps separating $\boldsymbol{x}$ and y in a greedy schedule. Let $slope(\boldsymbol{x}) = k_{\boldsymbol{x}}/p_{\boldsymbol{x}}$ be the maximum slope of any cycle on which $\boldsymbol{x}$ depends. If $\boldsymbol{x}$ is not dependent on any cycle, then $slope(\boldsymbol{x}) = 0/1$. We show that, in a greedy schedule, after scheduling $O(n^2)$ iterations, any subsequent occurrences

of a statement x are scheduled exactly k_x steps after the occurrence of x p_x iterations before. Thus, a pattern for each statement can be inferred from a greedy schedule of at most $O(n^2)$ iterations. The following lemmas are required to prove this theorem.

Lemma 4.4 Let C be a chain with $|C| \geq (i+1)n$ for some positive i. Then there are at least i disjoint subchains of C that are cycles.

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

Lemma 4.5 Given p integers $a_1 \ldots a_p$, then there is a subset S of the a_i such that

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

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

Theorem 4.6 Let x be a statement with slope k/p, and let the loop body contain n statements. In a greedy schedule, in any iteration i greater than $2np + 4p$, $dist(x_{i-p}, x_i) = k$.

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

- $Span(C) \leq np + p$ iterations. Let C' be a chain reaching x_{i-p}. Because dependencies are regular, a chain of dependencies identical to C reaches x_{i-p}. But $|C'| + k \leq |C|$, a contradiction.
- $Span(C) > np + p$ iterations. By Lemma 4.4, there are at least p disjoint subchains of C that are cycles. By Lemma 4.5, there is a subset of these cycles $\{C_h\}$ such that $\sum_h |C_h| = jp$ for some $j > 0$. Deleting the cycles $\{C_h\}$ from C produces a chain C' which reaches x_y, where $y = i - jp$. By assumption, there is a chain of length jk from x_y to x_i. But $\sum_h |C_h| \leq jk$, or else some C_h has slope greater than k/p, a contradiction. Therefore $|C'| + jk \geq |C|$, implying that $dist(x_y, x_i) = jk$. Since $dist(x_y, x_{i-p}) \geq (j-1)k$, $dist(x_{i-p}, x_i) \leq k$.

□

Theorem 4.6 shows that after scheduling $O(np_x)$ iterations, occurrences of x are scheduled at a constant (and optimal) rate. Thus, a sufficiently large greedy schedule exhibits a pattern for each statement.

Corollary 4.7 After scheduling $O(n^2)$ iterations every statement is scheduled at the optimal rate. Furthermore, if $p_x \leq 1$ for all x, then $O(n)$ iterations are sufficient.

Corollary 4.7 follows from Theorem 4.6 since $p_x \leq n$ for any x. The special case where $p \leq 1$ is important in practice: we have examined the inner loops of the Eispack routines [18] and the Livermore Loops [15] and have not found a single example where $p > 1$.

The efficiency of our algorithm is dependent on the cost of computing a greedy schedule. This is easily done using a modified topological sort of the dependency graph. The cost is proportional to the number of statements scheduled; thus, to schedule $O(n^2)$ iterations of n statements requires $O(n^3)$ time; if $p \leq 1$ then only $O(n)$ iterations must be scheduled, requiring $O(n^2)$ time.

5 Computing an Overall Pattern

We illustrate Theorem 4.6 with an example due to Cytron [8]. The loop is shown in Figure 2; the dependency graph is given in Figure 3a. Figure 3b shows the greedy schedule of five iterations. Iterations have been listed separately, side by side. The vertical axis is time; all statements on a horizontal line of the figure are executed simultaneously. For simplicity, the loop control code has been omitted.

```
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];
```

Figure 2: A sample loop.

In this example, the cycle B, E, H has the greatest slope (three). Statements C, D, and G are dependent on this cycle and thus have the same slope. All other statements have slope 0/1. In Figure 3b, the code is split into two groups that repeat every iteration, one with a slope of three, the other with a slope of zero.

There are two drawbacks to the simple greedy scheduling algorithm. First, to be assured that the pattern has been detected, it is necessary to run for $O(n^3)$ time. As observed above, in practice the pattern emerges much earlier, in $O(n^2)$ time or less. Second, the information greedy scheduling provides is not immediately useful for generating practical code. On a synchronous multiprocessor, each processor could be assigned a single statement x, which it would execute every p_x steps. (A minor modification allows statments with slope 0/1 to be handled smoothly.) For example, using the information in Figure 3b, statement G could be assigned to a processor which would execute the occurrence of G in iteration i at time $3i$. This is terribly inefficient, requiring $O(n^2)$ processors in the worst case.

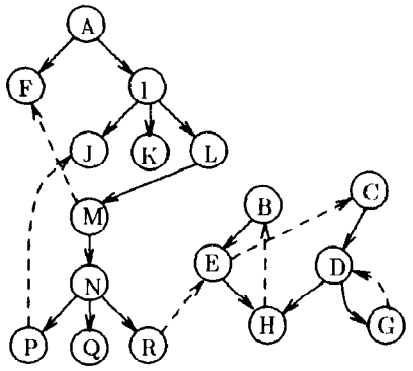
In this section we present a modification of greedy scheduling that detects a pattern for the entire loop body as soon as possible. The resulting code is much more efficient, using at most $O(n)$ processors. The idea is to reschedule statements not on the critical path so that they have the same slope as statements on the critical path. This results in a very compact pattern for the entire loop.

In Figure 3b, note that the statements with slope 0/1 in iterations four and five could be delayed without affecting the length of the schedule. Eliminating the "gaps" in the iterations—intervals of time steps with no statements from that iteration—produces the schedule in Figure 4. The boxed area is the pattern for the loop; scheduling additional iterations (without gaps) reproduces these three time steps. Note that no statement on the critical path has been delayed as a result of rescheduling statements. In what follows, we derive a method for determining when statements can be delayed without affecting the optimality of the final schedule.

When an iteration is scheduled, it is spread across some interval of time steps $t_1..t_k$. As in Figure 3b, statements from an iteration tend to cluster into groups of mutually dependent statements with gaps between the groups.

Definition 5.1 A *region* of a scheduled iteration is an interval of time steps $A = t_1..t_k$ such that some statement from the iteration is scheduled at every t_j.

We are interested in the maximal regions of an iteration. Between any two adjacent maximal regions there must be at least one step with no statement from that iteration. These gaps play a critical role in detecting a pattern.



(a) The dependency graph.

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

(b) The code after five phases.

Figure 3: Greedy scheduling.

In Figure 3b, there are two maximal regions for the last iteration: times 1..7 and 13..16.

Definition 5.2 Let $A_1, \ldots, A_j$ be the maximal regions of an iteration, where $A_i = t_i..t_i'$ and $t_i' < t_{i+1}$ for all i. Then $gap(A_i, A_{i+1}) = t_{i+1} - t_i'$.

In Figure 3b, iterations four and five have the same maximal regions; the only difference between the iterations is the size of the gap. In this sense, iteration five is a "stretched-out" version of iteration four. We say that two scheduled iterations i and $i+c$ are *alike* if they have the same maximal regions and the inter-region gaps in iteration $i + c$ are as large or larger than in iteration i. We present conditions under which the gaps in the larger iteration can be shrunk to the size of the gaps in the smaller iteration without affecting the optimality of the final code.

Theorem 5.3 Consider $i + c$ scheduled iterations, where iteration i and $i+c$ are alike, with j maximal regions. Assume there is no unbroken chain of dependent statements from a statement in A_k of iteration i ($k < j$) to a statement in A_j of iteration $i + c$. Then the inter-region gaps in iteration $i+c$ can be reduced to the size of the corresponding gaps in iteration i and for any additional (greedily) scheduled iterations the resulting schedule is optimal.

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

In Figure 3b, iterations four and five are alike and no chain from the first region of iteration four reaches to the second region of iteration five. Thus it is safe to shrink the gap in iteration five to the size of the gap in iteration four. There is also a simple test to determine when it is safe to completely close the gaps between regions; in this case, the gaps can be eliminated.[2] Figure 4 shows the schedule of Figure 3b extended with a few additional iter-

[2]It is not always safe to completely close the gaps in an iteration; examples can be exhibited where no scheduling strategy that does not introduce gaps can achieve an optimal schedule.

	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

Figure 4: The pattern.

ations and with the gaps completely closed.

The conditions of Theorem 5.3 require a check that iterations i and $i + c$ are alike, and that no chain of dependencies from a region in iteration i (except the last) reaches the last region of iteration $i + c$. For efficiency, we would like to minimize c, because implementing the dependency chain check is expensive relative to the cost of greedy scheduling. It can be shown that if $p_x \leq 1$ for all statements x, then checking consecutive iterations ($c = 1$) is sufficient. Checking if consecutive iterations are alike can be implemented without increasing the asymptotic complexity of the algorithm. Because $p \leq 1$ in practice, this yields an algorithm that runs in $O(n^2)$ in most cases.

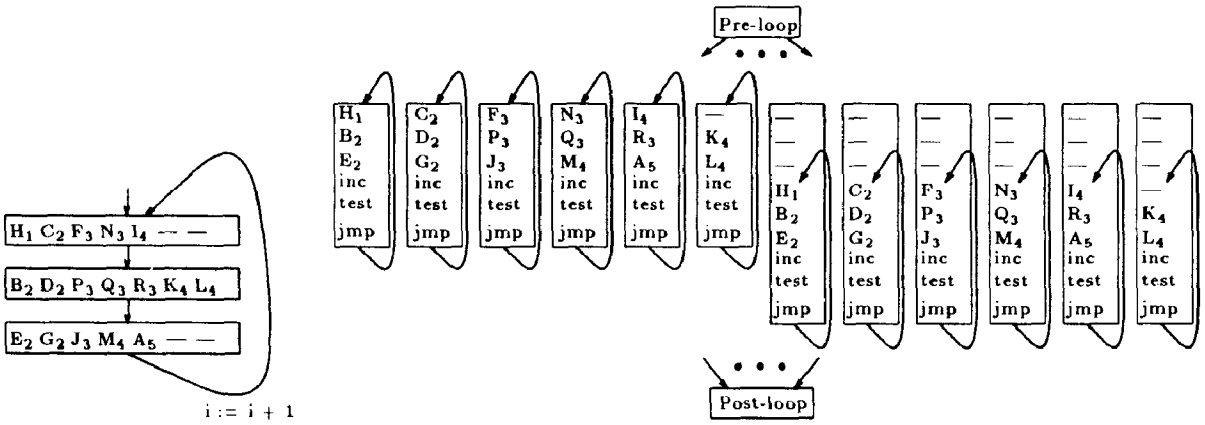
In theory, there are loops for which this strategy of making iterations "look alike" cannot succeed in polynomial time. Let the denominator of *slope*(x) be the *period* of x. The length of a pattern based on this approach is at least the least common multiple of the statement periods. If many statements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

6 Mapping Optimal Schedules to Processors

The transformed loop consists of a pre-loop (everything before the pattern), the pattern (with a backedge from the last step in the pattern to the first step in the pattern), and a post-loop (everything after the pattern). The final loop for the example is shown in Figure 5a.[3] The subscripts indicate the increment to the induction variable i.

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

[3]Including the postloop requires some adjustment of the loop bounds. This is easily computed for DO loops.



(a) The final program-graph. (b) Schedule for a synchronous machine.

Figure 5: An optimal schedule.

width of the pattern discovered by our algorithm, then the loop can be run in this raw form. Simple heuristics can reduce the width of the pattern without increasing its length. For example, in Figure 3a statement L has no dependents, and so can be delayed until the third node of the loop in Figure 5a, reducing the width from seven to six.

If the target machine is a wide-word architecture with a single flow of control, then the final program-graph can run directly on the machine, subject to including the loop overhead. If the target machine is a synchronous multiprocessor, the transformed loop can be vertically "sliced", with one statement from each node assigned to a processor. If the machine is not synchronous, then the exact strategy for code generation is heavily dependent on the machine's topology and the cost of communication. In general, though, the critical cycles of dependencies should be scheduled entirely on a single processor, with dependent statements scheduled on neighboring processors.

The only remaining detail is to include the loop overhead—the statements to increment the loop induction variable, to test for exit, and to jump. We outline a scheme for including the loop overhead on synchronous multiprocessors. The loop overhead is duplicated at the end of the loop body assigned to each processor. Each processor keeps its own copy of the induction variable in local storage, so the overhead computation is completely independent of any other processor. These statements can be masked by pipelining multiple occurrences of the pattern on different sets of processors—while one group is computing part of the pattern, another is computing loop overhead. Assuming that the overhead consists of an increment, test, and jump, an optimal schedule for the example for a synchronous multiprocessor is given in Figure 5b.

7 Experiments

We have implemented our scheduling algorithm, taking into account instruction latencies and processor limitations. Table 1 shows performance measurements for fourteen Livermore Loops [15]. The results are divided into three sections. Column two gives the flop rate of the original code on one pipelined processor, without statement reordering. Columns three and four give the flop rate of the schedules computed by our algorithm for limited resources of one and two pipelined processors, respectively. Columns five through seven show the processor requirements, register requirements, and flop rate of our algorithm's ideal schedules. We assume a machine of pipelined processors, each of which can initiate one instruction per cycle. Cray-1 instruction timings are used.

As shown in Section 5, our technique produces optimal schedules with respect to data dependencies. The speedup achieved by our algorithm, however, can depend on the hardware

and the extent to which the original code is optimized. In particular, the hardware support for indirect addressing used by vector machines (auto-increment of index registers) and a relatively sophisticated compiler optimization (removing redundant loads across iterations [7]) improve the speedups. To reflect this, some entries in Table 1 give a range. Standard compiler optimizations and 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; in LL6 redundant load removal greatly improved the performance of the original code.

The register and processor figures for the ideal schedules are upper bounds on the resources needed to achieve the flop rate in the last column. Some of the loops have no loop-carried dependencies and thus do not constrain parallelization. We have chosen to limit the ideal schedules of these loops to the minimum parallelism at which the sustained computation rate is one iteration per machine cycle.

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 is assumed.

No attempt is made to heuristically improve the match of the schedules computed by our algorithm to the resources—instructions are simply issued in the order of appearance in the pattern. Thus, these numbers are a lower bound on the performance achievable with our method.

The 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.

8 Conclusion

We have presented an algorithm to compute time-optimal schedules for doacross loops for synchronous parallel machines. If constraints are added, such as limited number of processors, asynchronous machines, expensive communication, or non-uniform memory access costs, then the general problem becomes NP-hard. However, we believe that even in these cases our algorithm is useful as a step to generating good parallel code. By starting with the maximum parallelism in a loop, good heuristic schedules can be obtained.

9 Acknowledgements

We would like to thank Dexter Kozen for the proof of Lemma 4.5. Laurie Hendren and Jennifer Widom criticized earlier drafts of this paper.

References

[1] AIKEN, A., AND NICOLAU, A. Loop Quantization: an analysis and algorithm. Tech. Rep. 87-821, Cornell Univ., 1987.

[2] AIKEN, A., AND NICOLAU, A. A development environment for horizontal microcode. *IEEE Trans. Softw. Eng.* (May 1988). Also available as Cornell Tech. Rep. TR 86-785.

[3] AIKEN, A., AND NICOLAU, A. Perfect Pipelining: A new loop parallelization technique. In *Proc. of the 1988 European Symp. on Programming* (Mar. 1988), Springer Verlag Lecture Notes in Computer Science. Also available as Cornell Tech. Rep. TR 87-873.

[4] ALLEN, J. R., AND KENNEDY, K. Automatic loop interchange. In *Proc. of the 1984 SIGPLAN Symp. on Compiler Construction* (June 1984), pp. 233-246.

[5] ALLEN, J. R., KENNEDY, K., PORTERFIELD, C., AND WARREN, J. Conversion of control dependence to data dependence. In *Proc. of the 1983 Symp. on Principles of Programming Languages* (Jan. 1983), pp. 177-189.

[6] C. POLYCHRONOPOULOS, D. KUCK, D. P. Execution of parallel loops on parallel processor systems. In *Proc. of the 1986*

Loop	Original Code	Limited Processors		Ideal Schedule		
	Mflops	1 proc Mflops	2 procs Mflops	procs	registers	Mflops
LL1	9	31-50	57-100	13	82	400
LL2	8	20-35	40-60	16	105	320
LL3	7	16-20	20-23	5	8	27-40
LL4	6	16	20	5	8	27
LL5	6	12-15	15-16	3	5	16
LL6	8	6-16	6-20	5	8	6-27
LL7	20	36-51	71-99	36	243	1280
LL8	11	40-55	80-110	60	363	2400
LL9	17	35-49	68-97	39	264	1360
LL10	10	18-25	36-48	40	210	720
LL11	4	4-9	4-11	4	4	4-13
LL12	4	13-20	27-40	6	37	80
LL13	4	11-12	22-24	50	376	560
LL14 (avg)	4	14-18	25-31	28	161	270
Average	8	19-28	35-50			534-537
Harmonic Mean	7	13-20	18-33			25-53

Table 1: Performance ranges for 14 Livermore Loops.

Int'l Conf. on Parallel Processing (Aug. 1986), pp. 519–27.

[7] CALLAHAN, D., COCKE, J., AND KENNEDY, K. Estimating interlock and improving balance for pipelined architectures. In *Proc. of the 1987 Int'l Conf. on Parallel Processing* (Aug. 1987), pp. 297–304.

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

[9] CYTRON, R. Doacross: Beyond vectorization for multiprocessors. In *Proc. of the 1986 Int'l Conf. on Parallel Processing* (Aug. 1986), pp. 836–844.

[10] EBCIOĞLU, K. A compilation technique for software pipelining of loops with conditional jumps. In *Proc. of the 20th Annual Workshop on Microprogramming* (Dec. 1987), pp. 69–79.

[11] FISHER, J. A. Trace Scheduling: A technique for global microcode compaction. *IEEE Trans. Comput. C-30*, 7 (July 1981), 478–90.

[12] GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[13] KUCK, D., KUHN, R., PADUA, D., LEASURE, B., AND WOLFE, M. Dependence graphs and compiler optimizations. In *Proc. of the 1981 Symp. on Principles of Programming Languages* (Jan. 1981), pp. 207–218.

[14] LAM, M. *A Systolic Array Optimizing Compiler*. PhD thesis, Carnegie Mellon Univ., 1987.

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

[16] MUNSHI, A., AND SIMONS, B. Scheduling sequential loops on parallel processors. Tech. Rep. 5546, IBM, 1987.

[17] PADUA-HAIEK, D. A. *Multiprocessors: Discussion of Some Theoretical and Practical Problems*. PhD thesis, Univ. of Illinois at Urbana-Champaign, 1979.

[18] SMITH, B. T., BOYLE, J. M., IKEBE, Y., KLEMA, V. C., AND MOLER, C. B. *Matrix Eigensystem Routines: EISPACK Guide*, 2 ed. Springer-Verlag, New York, 1970.