Perfect Pipelining:
A New Loop Parallelization Technique*

Alexander Aiken
Alexandru Nicolau

87-873

October 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

*This work was supported in part by NSF grant DCR-8502884 and the Cornell NSF Supercomputing Center.

Perfect Pipelining: A New Loop Parallelization Technique[1]

Alexander Aiken
Alexandru Nicolau

Computer Science Department
Cornell University
Ithaca, New York 14853

[1]This work was supported in part by NSF grant DCR-8502884 and the Cornell NSF Supercomputing Center.

Abstract

Parallelizing compilers do not handle loops in a satisfactory manner. Fine-grain transformations capture irregular parallelism inside a loop body not amenable to coarser approaches but have limited ability to exploit parallelism across iterations. Coarser methods sacrifice irregular forms of parallelism in favor of pipelining (overlapping) iterations. In this paper we present a new transformation, *Perfect Pipelining*, that bridges the gap between these fine- and coarse-grain transformations while retaining the desirable features of both. This is accomplished even in the presence of conditional branches and resource constraints. For loops typically encountered in practice, Perfect Pipelining achieves the effect of full loop unrolling coupled with fine-grain parallelization. To make our claims rigorous, we develop a formalism for parallelization. The formalism can also be used to compare transformations across computational models. As an illustration, we show that *Doacross*, a transformation intended for synchronous and asynchronous multiprocessors, can be expressed as a restriction of Perfect Pipelining.

1 Introduction

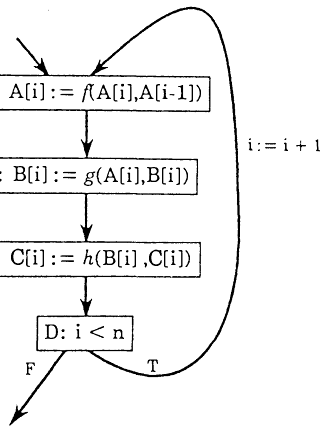
A significant amount of research has been done in the area of parallelization, the extraction of parallelism from sequential programs. The extraction of *fine-grain parallelism*—parallelism at the level of individual instructions—using code compaction has recently emerged as an important sub-field. The model of computation for compaction-based parallelization is generally some form of shared-memory parallel computer consisting of many synchronous, statically-scheduled functional units with a single flow of control. Programs for these machines may be depicted as program graphs where nodes can contain multiple operations. Transformations on these programs rearrange operations to shorten—compact—the paths through the program graph. Numerous commercial machines (including Multiflow's Trace series, CHOPP, Cydrome, the FPS series, horizontal microengines, and RISC machines) use compaction techniques to exploit parallelism.

The standard approach to extracting parallelism from a loop through compaction is to compact the loop body. This yields some performance improvement, but does not exploit parallelism that may be present between separate iterations of a loop. To alleviate this problem, most systems *unroll* (replicate) the loop body a number of times before compacting. If a loop is unrolled k times, parallelism can be exploited inside this unrolled loop body, but the new loop still imposes sequentiality between every group of k iterations. We present a new loop parallelization technique, *Perfect Pipelining*, that overcomes this problem by achieving the effect of unbounded unrolling and compaction of a loop.

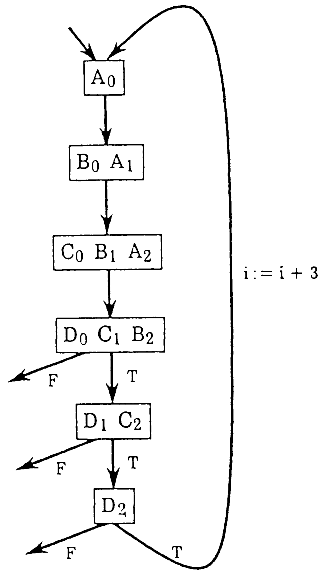
To illustrate the importance of Perfect Pipelining, consider the program graph shown in Figure 1a. (We have simplified the loop control code for clarity: the induction variable i is incremented implicitly on the backedge, as in a Fortran DO loop.) The running time of this loop is $4n$ steps, where n is the number of iterations executed. Multiple iterations of this loop may be overlapped subject to the constraint that the first operation of an iteration is dependent on the result of the first operation of the previous iteration. Figure 1b shows a schedule after the loop has been unwound two times and compacted.[1] Operation labels have been substituted for the operations; subscripts indicate the increment to the induction variable. Multiple operations within a node are evaluated concurrently. The running time of this loop is $2n$ steps. Figure 1c shows the loop unwound four times and compacted; in this case the running time is $\frac{8}{5}n$ steps. Note the low parallelism at the beginning and end of the loop body in both of these examples.

Additional unrolling and compaction will improve the running time further, although

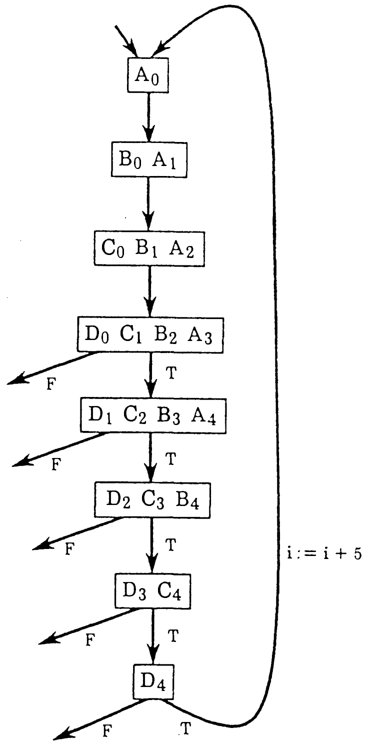
[1]Two additional memory locations are allocated to each array to handle the extra references generated when $i = n$.



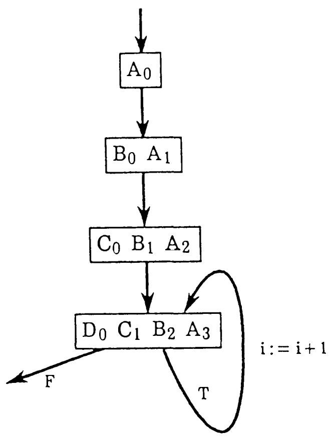
(a) Original loop.



(b) Unwound two times and compacted.



(c) Unwound four times and compacted.



(d) Loop after pipelining.

Figure 1: A Perfect Pipelining example.

this becomes computationally expensive very rapidly. Existing compaction transformations can achieve the schedules in Figures 1b and 1c. Perfect Pipelining derives the program shown in Figure 1d. Intuitively, the transformation accomplishes this by noticing that the fourth and fifth nodes of Figure 1c execute the same operations from different iterations, and that further unrolling and compaction will create more nodes of the same type. The transformation achieves continuous (or perfect) pipelining of the loop iterations. The running time for this loop is $n + 3$ steps.

In the example, the pattern detected by Perfect Pipelining is very simple because there are no branches (other than exits) in the loop body. A surprising property of Perfect Pipelining is that it finds such a pattern on all paths given arbitrary flow of control within the loop body. This is a substantial improvement over previous techniques, which rely on heuristics to estimate the runtime flow of control [Fis81] or ignore branches altogether. Another important property of Perfect Pipelining is that the transformation applies even in the presence of resource constraints. We prove that the transformation finds a pattern given arbitrary resources and provide an example illustrating its performance when the loop has unpredictable flow of control and machine resources are a limiting factor.

Perfect Pipelining is defined using the primitive transformations of *Percolation Scheduling* [Nic85b] and loop unrolling. Percolation Scheduling is attractive because the transformations are efficient and simple, involving only a single instruction and adjacent nodes of the program graph. In this paper, we briefly describe each of these primitive transformations.

To make our claims precise, we develop a formal account of our transformations. We define the language to which the transformations apply and provide an operational semantics. A binary relation $\leq_p$ is defined on programs using the operational semantics. If $P \leq_p P'$ then operations in P' are executed earlier than corresponding operations in P. In this sense, $\leq_p$ measures when one program is "more parallel" than another. We use $\leq_p$ to prove that Perfect Pipelining is better than any finite unrolling with compaction.

The formalism requires a fair amount of development, but the result is powerful enough to capture the intuitive notion of program improvement used informally throughout the literature on parallelization. Thus, we can use $\leq_p$ to compare seemingly unrelated transformations in a meaningful way. As an example, we show that the *Doacross* transformation [Cyt86] can be derived as a restriction of Perfect Pipelining. Since Doacross is a loop pipelining transformation intended for synchronous or asynchronous (loosely-coupled) multiprocessors, this result suggests that our formalism is generally applicable across the various models of computation and transformations proposed in the field of program parallelization.

2 A Simple Language

In this section we give an informal description of SPL, a Simple Parallel Language. In the next section we develop a formal definition of the language and an operational semantics. We have minimized the details of language design while keeping the language rich enough to allow discussion of the important problems. SPL is not so much a "real" programming language as a tool convenient for discussing parallelizing transformations.

SPL is graphical; programs are represented by a control flow graph as in Figure 1a. Each node in the graph contains zero or more primitive operations. These operations are divided into two categories: assignments and tests. The evaluation of an assignment updates the store, while tests affect the flow of control. Execution begins at the start node and proceeds sequentially from node to node. When control reaches a particular node, all operations in that node are evaluated concurrently; the assignments update the store and the tests return the next node in the execution sequence (see discussion below). Operations evaluated in parallel perform all reads before any assignment performs a write. Write conflicts within a node are not permitted.

Care must be taken to define how multiple tests are evaluated in parallel. The set of tests within a node is given as a directed acyclic graph (*dag*). Each test in the dag has two successors corresponding to its true and false branches. A successor of a test is either another test or a *name*; a name is a pointer to a program node. We require that the dag of tests be *rooted*—that it have a single element with no predecessors. To evaluate a dag in a state, select the (unique) path from the root to a name such that the branches on the path correspond to the value (true or false) of the corresponding test in the state. Evaluation of the dag returns the node name that terminates this path.

On a real machine the evaluation of multiple tests can be very sophisticated to exploit parallelism. For example, the tests can be evaluated simultaneously and a fast algorithm used to select the correct node name from the boolean results. A hardware mechanism that efficiently implements general dags of tests is described in [KN85]; less general multiway jump mechanisms are used in many horizontal microengines and the Multiflow architecture.

SPL is powerful enough to model execution of a tightly-coupled parallel machine at the instruction level. It is at this level that our transformational system extracts parallelism from programs. A sample SPL program is shown in Figure 1a. Note that this program has only one operation per node; such a program is *sequential*. Another, more parallel version of the same program is given in Figure 1b.

$$\begin{array}{lcl} \text{Bool} & = & \text{tt} + \text{ff} \\ \text{Loc} & = & \mathcal{Z} \\ \text{Store} & = & \text{Loc} \rightarrow \text{Val} \\ \text{Assign} & = & \text{Store} \rightarrow \text{Loc} \times \text{Val} \\ \text{Test} & = & \text{Store} \rightarrow \text{Bool} \end{array}$$

Figure 2: Basic domains.

3 Language Definition and Operational Semantics

The formal definition of SPL and its operational semantics provide a framework for proving properties of program transformations. In subsequent sections we develop a formalism for our transformations; this formalism uses the operational semantics of SPL to define when one program is more parallel than another. The operational semantics of SPL closely follows the structural style advocated by Plotkin [Plo].

Figure 2 lists the basic domains of SPL. Val is a domain of basic values—integers, floating-point numbers, etc. An assignment, a function of type Assign, deviates from the standard approach in that it does not return an updated store. Instead, an assignment returns a pair $\langle l, v \rangle$, where v is the new value of location l. This allows us to define the parallel execution of several assignments as the parallel binding of the new values to the updated locations. A *program* is a tuple $\langle N, n_0, F \rangle$ where:

- N is a finite set of *nodes*
- $n_0 \in N$ is the start node
- $F \subseteq N$ is the set of final nodes

A node is a pair $\langle A, C \rangle$ where:

- A is a set of *assignments*
- C is a *dag*; a four-tuple $\langle B, \mathit{select}, r, H \rangle$ where:
  - B is a set of *tests*
  - $\mathit{select} : B \times \text{Bool} \rightarrow B + H$ is an edge function
  - r is the root test or a node name
  - H is a set of node names

In what follows, s and s' range over stores; variants of v, l, a, and t range over values, locations, assignments, and tests respectively. We assume that assignments and tests are total atomic actions of type Assign or Test. We use n for both the name of a node and the node itself; the meaning is always clear from the context.

The transformations we define require knowledge of the locations that are read and

Let $n = \langle A, C \rangle$, $C = \langle B, select, r, H \rangle$, and $A = \{a_i\}$

Constraints on C:

The directed graph induced on B and H by *select* is:

connected

acyclic

has a unique root r.

Constraint on A:

$\forall s, i, j \; write(a_i, s) \cap write(a_j, s) = \emptyset$

Constraints on n:

$\forall n \in N \; n_0 \notin succ(n)$

$\forall n \in F \; succ(n) = \emptyset$

$\forall n = \langle A, \langle B, select, r, H \rangle \rangle \in F \; A = \{result\} \wedge B = \emptyset$

Figure 3: Program constraints.

written by the primitive operations to model *dependency analysis*. Dependency analysis determines when two program statements may refer to the same memory location. The analysis is used to determine when it is safe to perform instructions in parallel. We define $write(a, s)$ to be the location written by assignment a in store s; $read(a, s)$ is the set of locations read by assignment (or test) a in store s.

In Section 2, we discussed well-formedness conditions and semantic constraints on programs that are not implemented by the above description. These requirements are listed in Figure 3. The constraints ensure that the dag of tests is well-formed and that two assignments in a node cannot write the same location. In addition, the start node should have no predecessors and a final node should have no successors. A final node contains a distinguished operation, *result*, that reads and returns the result of the computation. For the purposes of this paper, we assume that *result* returns the entire final store.

Figure 4 gives an operational semantics for SPL. The semantics consists of a set of rewriting rules in the style of inference rules of formal logic. There are two types of transitions: $\leadsto$, which defines transitions within a node, and $\longrightarrow$, which defines transitions between nodes. Rules are read as stating that the assertion below the line holds if the assertions above the line hold. The first two rules deal with the evaluation of a dag of tests; the third rule describes the parallel evaluation of assignments. The fourth rule defines the execution of a node in terms of the evaluation of the node's test dag and assignments.

A rewriting sequence is an execution history of one computation of a program. For

$$\frac{C = \langle B, select, r, H\rangle,\ t \in B,\ select(t, t(s)) = t'}{\langle C, s, t\rangle \rightsquigarrow \langle C, s, t'\rangle}$$

$$\frac{C = \langle B, select, r, H\rangle,\ n' \in H}{\langle C, s, n'\rangle \rightsquigarrow n'}$$

$$\frac{A = \{a_i\}, a_i(s) = \langle l_i, v_i\rangle,\ s[\ldots, l_i \leftarrow v_i, \ldots] = s'}{\langle A, s\rangle \rightsquigarrow s'}$$

$$\frac{n = \langle A, C\rangle, C = \langle B, select, r, H\rangle,\ n \notin F,\ \langle C, s, r\rangle \overset{*}{\rightsquigarrow} n',\ \langle A, s\rangle \rightsquigarrow s'}{\langle n, s\rangle \rightarrow \langle n', s'\rangle}$$

Figure 4: Operational semantics of SPL.

our purposes, a complete sequence contains much irrelevant detail; in particular, we are rarely interested in the internal evaluation of a node (the $\rightsquigarrow$ transitions). The following definition puts a rewriting sequence at the right level of abstraction for viewing execution as transitions from nodes to nodes:

Definition 3.1 The *execution trace* of program P in initial store s, written $T(P, s)$, is the sequence $\langle n_0, s_0\rangle \rightarrow \langle n_1, s_1\rangle \rightarrow \langle n_2, s_2\rangle \rightarrow \ldots \rightarrow \langle n_k, s_k\rangle$ where $s_0 = s$, n_0 is the start node of P, and $n_k \in F$. Traces are defined only for terminating computations.

4 The Core Transformations

The core transformations are the building blocks of Perfect Pipelining. These primitive transformations are local, involving only adjacent nodes of the program graph. Though simple, the core transformations can be used to express very powerful code motions [AN88].

Definition 4.1 The result $R(P, s)$ of a computation is the final store of $T(P, s)$. Two programs P and P' are *strongly equivalent* if $\forall s\ R(P, s) = s' \Leftrightarrow R(P', s) = s'$.

We require that a transformation of a program be strongly equivalent to the original. This is formalized as a correctness condition. This guarantees that the result of any sequence of transformations is strongly equivalent to the original program.

succ: Node $\rightarrow \mathcal{P}$(Node)
$succ(n) = H$ where $n = \langle A, \langle B, select, r, H \rangle\rangle$

pred: Node $\rightarrow \mathcal{P}$(Node)
$pred(n) = \{n' | n \in succ(n')\}$

op: Node $\rightarrow \mathcal{P}$(Assign + Test)
$op(n) = A \oplus B$ where $n = \langle A, \langle B, select, r, H \rangle\rangle$

node: Assign + Test $\rightarrow$ Node
$node(x) = n$ where $x \in op(n)$

Figure 5: Some useful definitions.

Definition 4.2 (Correctness) Let $\mathcal{T}$: Pgm $\rightarrow$ Pgm be a program transformation. Then $\mathcal{T}$ is *correct* if for all P, $\mathcal{T}(P)$ is strongly equivalent to P.

The formal definitions of the transformations and proofs of correctness can be found in the appendix. In this section, we briefly describe and illustrate each transformation.

Figure 5 lists some useful functions. *Succ* returns the immediate successors of a node; when it is convenient we refer to an edge (m, n) instead of writing $n \in succ(m)$. *Pred* returns the immediate predecessors of a node. The function *op* returns the operations in a node. *Node*(x) is the node containing operation x (we assume there is some way of distinguishing between multiple copies of the same operation).

4.1 Delete

The *Delete* transformation removes a node from the program graph if it is empty (contains no operations) or unreachable. A node may become empty or unreachable as a result of other transformations. Figure 6 gives a picture. Only the relevant portion of the program graph is shown; incoming edges are denoted by I_j and exiting edges by E_j. Note that an empty node has exactly one successor.

4.2 Move-op

The *Move-op* transformation moves an assignment x from a node n to a node m through an edge (m, n) provided no conflict exists between x and the operations in m and x does not kill any value live at m. Care must be taken not to affect the computation of paths

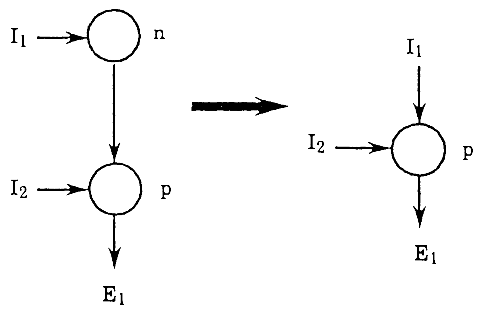


Figure 6: The delete transformation.

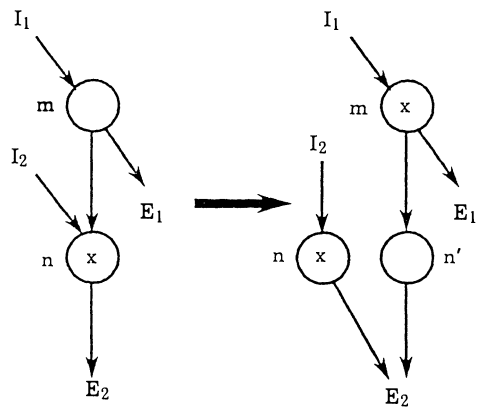


Figure 7: The move-op transformation.

passing through n but not through m. To ensure this, the original node n is preserved on all other paths. A picture is given in Figure 7. Node n' is n with assignment x removed.

4.3 Unify

The *Unify* transformation moves a single copy x of identical assignments from a set of nodes $\{n_j\}$ to a common predecessor node m.[2] This is done if no dependency exists between x and the operations of m and x does not kill any value live at m. Paths passing through an n_j but not through m must not be affected—as in Move-op, the original nodes are preserved on other paths. An illustration is given in Figure 8.

[2]Unify can be defined to allow distinct operations that are mutually dependent to move into a node m [Nic85b], but we do not need this generality for Perfect Pipelining.

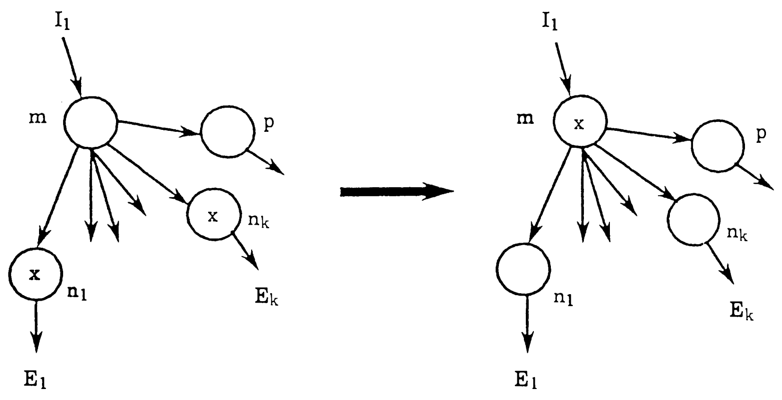


Figure 8: The unify transformation.

4.4 Move-test

The *Move-test* transformation moves a test x from a node n to a node m through an edge (m, n) provided that no dependency exists between x and the operations of m. Paths passing through n but not through m must not be affected; n is preserved on the other paths. Because we allow an arbitrary rooted dag of tests in a node and the test being moved may come from an arbitrary point in that dag, n is split into n_t and n_f, where n_t and n_f correspond to the true and false branches of x. An illustration of the transformation is given in Figure 9. In the illustration, a represents the dag of tests (in n) not reached by x, b represents the dag of tests reached on x's true branch, and c the dag of tests reached on x's false branch.

5 Unrolling

Loop *unrolling* (or unwinding) is a standard non-local transformation. When a loop is unrolled, the loop body is replicated to create a new loop. Loop unrolling helps exploit fine-grain parallelism by providing a large number of operations (the unrolled loop body) for scheduling. The operations in the unwound loop body come from previously separate iterations and are thus freer of the order imposed by the original loop. Recent work has focused on the correct unwinding of multiple nested loops [Nic85a,AN87,CCK87]. An illustration of a loop unrolled once is given in Figure 10. The formal definition of unrolling is included in the appendix; we use the shorthand u^iL for loop L unrolled i times.

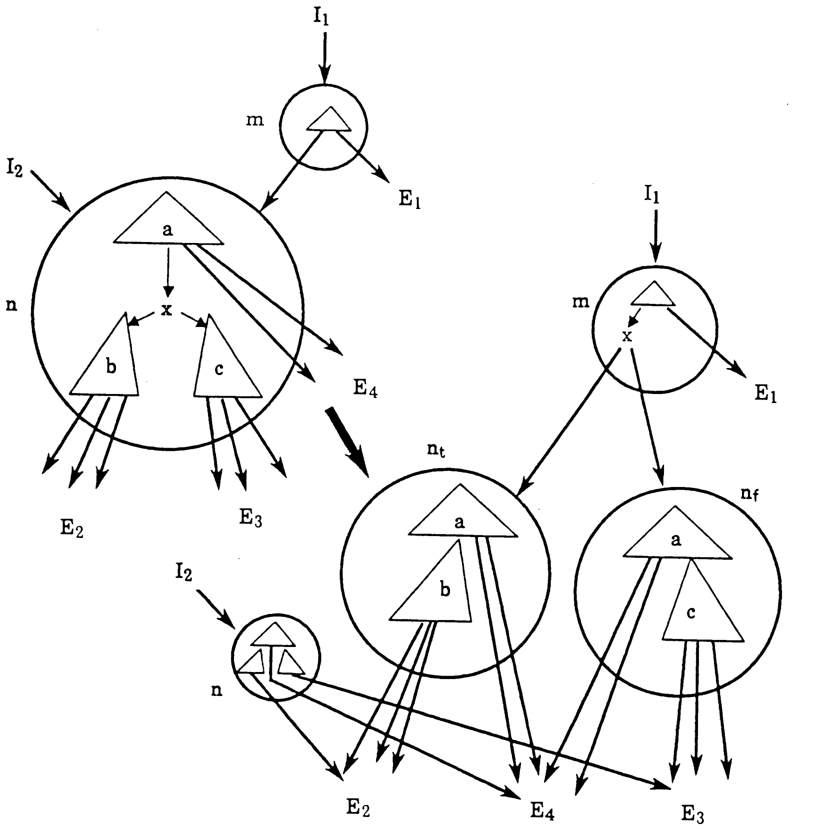


Figure 8: The move-test transformation.

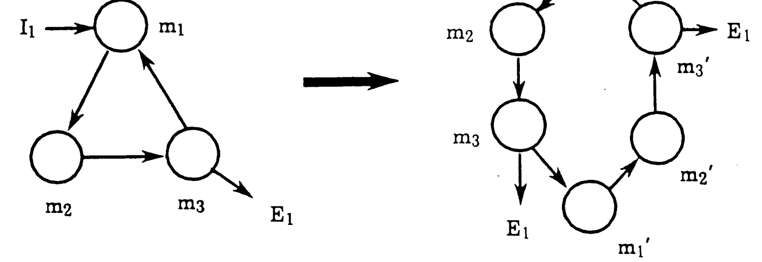


Figure 9: The unrolling transformation.

6 A Formalization of Parallelism

In this section we develop a formal account of our transformations. This will allow us to make precise claims about the effect of Perfect Pipelining and to compare Perfect Pipelining with other transformations. We restrict the development to transformations that exploit only control and dependency information; this is a natural and large class of transformations (including our transformations) dominating the literature on parallelization. Examples of transformations in this class include: vectorization, the hyper-plane method [Lam74], loop distribution [Kuc76], loop interchange [AK84], trace scheduling [FERN84], and Doacross [Cyt86].

We introduce a preorder on programs, "**sim**" (for similarity), that captures when one program approximates the control and dependency structure of another. We then introduce a partial order $\leq_p$ that is a restriction of **sim**. If $P \leq_p P'$, then P' is a more parallel program than P.

Informally, a program P is **sim** to P' if P' executes the same operations as P in an order compatible with the data and control dependencies present in P. P' may, however, have additional operations on some paths that do not affect the output of the program. The sample program in Figure 1b has more operations on some paths than the program in Figure 1a, but the two programs compute the same function. A transformation that converts P into P' may take advantage only of the control and data dependency information that can be inferred from P; other semantic properties of a program are not exploited. [3]

The purpose of **sim** is to establish a sensible correspondence between operations in traces of P and operations in traces of P'. It is not enough to simply ensure that copies of operations from P appear in P', since there is no guarantee that dependencies are preserved. The following development provides the machinery needed to establish this mapping.

Definition 6.1 We say that y *depends on* x in trace $T(P, s)$, written $x \prec y$, if y reads a value written by x. Formally, let $\langle n_0, s_0\rangle \xrightarrow{*} \langle n_i, s_i\rangle \xrightarrow{*} \langle n_j, s_j\rangle$. Then $x \prec y$ if $x \in op(n_i)$, $y \in op(n_j)$, $write(x, s_i) \subseteq read(y, s_j)$, and there is no operation z in n_k for $i < k < j$ such that $write(x, s_i) = write(z, s_k)$.

The relation $\prec$ models *true* dependencies [Kuc76], which correspond to actual definitions and uses of values during execution. This is not conservative dependency analysis—the

[3]There are techniques which make good use of other semantic properties of programs. For example, tree height reduction [KM75] uses the associativity of operations to balance the parse tree of an arithmetic expression. The expression can then be evaluated on a parallel machine in time proportional to the depth of the transformed parse tree. This method and others like it are not considered here, but are not incompatible with our transformations.

relation $\prec$ precisely captures the flow of values through an execution of a program.

Definition 6.2 Consider a trace $T(P, s)$. The *trace dependency graph* is $D = (V, E)$, where V is the set of all operations in the trace and there is an edge between every pair of dependent operations; i.e., $E = \{(x, y) | x \prec y\}$. Multiple copies of the same operation appearing in the trace are distinct nodes in the dependency graph.

The following definition allows us to concisely state when a program P' preserves the dependencies of P.

Definition 6.3 Given $D = (V, E)$, the trace dependency graph of $T(P, s)$, let $D_x = (V_x, E_x)$, the restriction of D to an operation x, be the subgraph induced by all operations on which x directly or indirectly depends. Defined inductively:

- $x \in V_x$
- $x \in V_x \wedge (y, x) \in E \Rightarrow y \in V_x \wedge (y, x) \in E_x$

There is a natural equivalence relation on operations in a trace—two operations x and y are in the same equivalence class if D_x is isomorphic to D_y. Two operations in the same class compute the same value. The equivalence class of an operation x is denoted by $\bar{x}$. Elements within a class are distinguished by subscripts indicating relative position in the trace—the ith occurrence in the trace of an operation in $\bar{x}$ is $\bar{x}_i$. If two operations are not syntactically equal, then they are not in the same equivalence class; however, it is certainly possible for two identical operations to have different dependency graphs and therefore be in different equivalence classes. For example, assume two copies x and x' of the statement $i \leftarrow i + 1$ appear in a trace. If one copy depends on the other, then D_x is not isomorphic to $D_{x'}$.

Although operations with isomorphic dependency graphs compute the same value, it is not safe to eliminate these "useless" operations from the program graph. Operations that appear redundant in one trace may be useful in another. For our notion of similarity, we require that the operations of P, useful or not, be present in the traces of P' if P **sim** P'. The formalism can be extended to handle the removal of dead code; however, we prefer to develop the ideas in the simpler setting

Definition 6.4 (Similarity)

$$P \textbf{ sim } P' \Leftrightarrow \forall s \; \bar{x}_i \prec \bar{y}_j \text{ in } T(P, s) \Rightarrow \bar{x}_{f_{\bar{x}}(i)} \prec \bar{y}_{f_{\bar{y}}(j)} \text{ in } T(P', s)$$

where $f_{\bar{o}}$ is a one to one function on the integers.

The functions $f_{\bar{o}}$ provide a mapping demonstrating that P' preserves the dependency structure of P. It can be shown that P is strongly equivalent to P' if P **sim** P'. We now introduce the relation $\leq_p$. If $P \leq_p P'$, then all operations in P' are executed at least as early in the trace as corresponding operations in P.

Definition 6.5 Let $x \in op(n_i)$ in $T(P, s)$. The position of x, written $pos(x)$, is i.

$$P \leq_p P' \Leftrightarrow P \textbf{ sim } P' \wedge \forall s \; pos(\bar{x}_i) \text{ in } T(P,s) \geq pos(\bar{x}_{f_x(i)}) \text{ in } T(P', s)$$

It can be shown that $\leq_p$ is a reflexive partial order. The following theorem justifies the core transformations by showing that some improvement results from their application.

Theorem 6.6 Let $\mathcal{T}$ be any core transformation or unrolling. Then for all P, $P \leq_p \mathcal{T}(P)$.

Proof: [sketch] The transformations preserve dependencies and do not remove an operation from any path on which it occurs—thus P **sim** $\mathcal{T}(P)$. For each core transformation, if it succeeds, at least one operation appears earlier on at least one path, so $P \leq_p \mathcal{T}(P)$. □

7 Pipelining Loop Iterations

Existing compaction systems all use the same technique to exploit parallelism across iterations of a loop. The loop is unwound a number of times and the new loop body is compacted. If there are no dependencies between the unwound iterations, then for a fixed size machine there is an unwinding that yields near optimal resource utilization after compaction.

If there are dependencies between the unwound iterations the result can be much worse. Typically, the compacted loop has nodes containing many operations near the beginning of the loop, but towards the end of the loop body operations "thin out" because of dependency chains between unwound iterations and the fact that there are no more operations to schedule. Thus the code becomes increasingly sequential towards the end of the compacted loop body. The problem can be somewhat alleviated by additional unwinding and compaction; however, this becomes computationally expensive very rapidly and there will still be a "tail" of sequential code at the end of the loop body. For example, the loop in Figure 1c shows the result of unwinding and compacting the loop in Figure 1a. Note the low parallelism at the beginning and end of the loop body.

We apply the results of the previous sections to develop a new loop transformation, *Perfect Pipelining*, that has the effect of unbounded unwinding and compaction. This transformation cannot be achieved directly using the core transformations. For this reason, the relation $\leq_p$ is crucial to proving properties of Perfect Pipelining.

7.1 The Problem

For simplicity, we disregard the particular strategy for compacting a loop and assume only that we are given a deterministic compaction operator $\mathcal{C}$ built on the core transformations. We assume that a program is a simple (innermost) loop of the type discussed in the section on unrolling. Nested loops can be handled using techniques for unrolling multiple loops [AN87].

Consider the sequence $\mathcal{C}u^0L, \mathcal{C}u^1L, \mathcal{C}u^2L, \ldots$. If $\forall i\ \mathcal{C}u^iL \leq_p \mathcal{C}u^{i+1}L$, then $\mathcal{C}$ is said to be *well-behaved.* We give a method, for a class of programs and well-behaved compaction operators, to compute a program $\mathcal{C}u^\infty L$ satisfying

$$\forall i\ \mathcal{C}u^iL \leq_p \mathcal{C}u^\infty L$$

7.2 The Programs

A loop u^iL consists of unwound iterations $L_0, \ldots, L_i$. A *loop carried dependency* [AK84] is a dependency between separate iterations of a loop. In this context we are referring to the approximate dependency graphs a compiler computes using conservative dependency analysis, rather than the precise trace dependency graphs used to define $\leq_p$. We consider single (non-nested) loops satisfying the following property for any unwinding:

Constraint 7.1 Assume there is a loop carried dependency between operation x and y in L. Then in u^iL, there is a dependency between operation x of L_j and y of L_{j+1} for all j.

Virtually all loops encountered in practice can be mechanically rewritten to satisfy this constraint [MS87]. In essence, the requirement is that the dependencies present in a loop unwound i times are a good predictor of the dependencies in the loop unwound $i+1$ times. In practice, these conditions can be checked by inspection of the loop without resorting to computation of the dependency graph.

8 Compaction Operators

We are interested in the class of *bounded* compaction operators. The key characteristic of these operators is that on any path of $\mathcal{C}u^iL$ the distance between the first and last scheduled operations of L_j is bounded by a constant. The fact that any iteration L_j cannot be "stretched" too much allows us to compute $\mathcal{C}u^\infty L$. We present the simplest bounded operator, the simple rule. More powerful bounded operators are discussed in the appendix. Initially we assume that computational resources are unlimited; in Section 10 we discuss Perfect Pipelining when resources are bounded.

```
move(x, n, m)
    if x is an assignment
        then P ← unify(P, x, n, m)
        else P ← move-test(P, x, n, m)
    if no change in P
        then return (False)
        else return (True)
```

Figure 11: The Move operator.

```
move_iteration(x, n, m)
    if x ∈ op(n)
        then
            if ¬move(x, n, m) then fail;
    (* next_op_in_it(x, n, p) is next operation in the iteration after x on edge (n, p). *)
    for each ⟨p, y⟩ such that p ∈ succ(n) ∧ next_op_in_it(x, n, p) = y do
        move_iteration(y, p, n);
    Delete all empty nodes;
```

Figure 12: Moving an iteration.

8.1 The Simple Rule

To simplify the algorithms, we combine the primitives Unify and Move-test into one transformation *Move* (see Figure 11). Move is not the best such transformation—it does not even use Move-op—but it is the simplest.

The simple rule moves an iteration L_j as far "up" in the program graph on as many paths as possible. Operations in the iteration remain in adjacent nodes and the iteration keeps its "shape"—operations appear in the order of the original loop body. These restrictions are not great; the original loop body L could have been compacted prior to application of unrolling and the simple rule, in which case the operations in an unwound L_j are actually nodes containing multiple operations.

The basic step of the simple rule is to move each operation in one copy of an iteration up one node in the program graph. An algorithm that accomplishes this is given in Figure 12. We assume that operations are identified with their L_j. A **fail** command causes the entire recursive computation to terminate and restores the original program graph.

(* Let $P = u^i L$ *)
for each iteration $L_0, \ldots, L_i$ **do**
    $X \leftarrow \{x\}$ where x is the first operation in L_j
    repeat
        (* we assume that X always contains all copies of operation x *)
        1. **while** $\exists y \in X$ s.t. $pred(node(y)) = \{p\}$ and y's iteration can move
            do $move\_iteration(y, node(y), p)$

        2. **if** $\exists y \in X$ s.t. y can move to node $p \in pred(node(y))$
            and the rest of the iteration can move accordingly **then**
                select y s.t. the depth—ignoring backedges—of $node(y)$
                    in the program graph is maximized;
                $move\_iteration(y, node(y), p)$
    until 2 fails.
    Delete all empty nodes.

Figure 13: The simple rule.

The simple rule is given in Figure 13.[4] The algorithm guarantees that all possible unifications are performed, thus minimizing code explosion. As iterations move through the program graph, copies of operations—forming distinct copies of the iteration—are generated where paths split. The top-level algorithm refers to the first operation in each copy of the iteration; the other operations are handled by Move_iteration. Let $\mathcal{C}$ stand for the simple rule. An important property of $\mathcal{C}$ is that it is maximal—for any $\mathcal{C}'$ using Move_iteration and for all programs P and unrollings i, $\mathcal{C}u^iP \not<_p \mathcal{C}'u^iP$. The simple rule is well-behaved.

Figure 1c shows a loop unwound and compacted using $\mathcal{C}$. The only loop carried dependency is between the first operation of consecutive iterations; after application of $\mathcal{C}$ the iterations overlap, staggered by one node.

9 Perfect Pipelining

In this section, we require that loop carried dependencies satisfy Constraint 7.1 and that there be enough such dependencies that $\mathcal{C}$ cannot completely overlap unwound iterations on any path. In Section 10 we show how this stronger condition can be removed; further dis-

[4]We have omitted one detail from the simple rule. Because tests are not unified, it is possible to have extra copies of a test in the program graph. The details of eliminating the redundant copies are in [AN88].

cussion of dependency constraints is included in the appendix. The following two properties of the simple rule are required for Perfect Pipelining.

Definition 9.1 Two nodes n and n' are *equivalent* if they have the same operations (from different iterations) and dag structure and there is a k such that if operation $x \in op(n)$ is from iteration L_j, then $x \in op(n')$ is from iteration L_{j+k}.

Lemma 9.2 (Property 1) Let n and n' be nodes in $\mathcal{C}u^iL$. Assume i is large enough that the successors of n and n' are unaffected by larger unwindings and applications of $\mathcal{C}$—the stronger dependency assumption guarantees the existence of i. If n and n' are equivalent, then corresponding successors of n and n' are equivalent.

Proof: [sketch] Let p and p' be corresponding successors of n and n' respectively. Because operations from the an iteration are adjacent and in the same order after the application of $\mathcal{C}$, it follows that x is an operation of iteration L_j in n and y is an operation in L_j in p if and only if x is an operation of L_{j+k} in n' and y is an operation of L_{j+k} in p'.

The only other possibility is that p contains the first operation of an iteration L_j. If there is a loop carried dependency, then, because loop carried dependencies are regular and $\mathcal{C}$ is maximal, a moving iteration is always be blocked by the same operation from the previous iteration on a particular path. Thus, the first operation of iteration L_j is in p if and only if the first operation of L_{j+k} is in p'. The fact that p and p' have the same dag structure follows from the definition of the simple rule and fact that n and n' have the same dag structure. □

Lemma 9.3 (Property 2) There is a constant c, dependent only on L, satisfying

$$\forall i \; n \in \mathcal{C}u^iL \Rightarrow |op(n)| < c$$

.

Proof: Because an iteration's operations are contiguous and no consecutive iterations overlap completely on any path, a node may have operations from a number of consecutive iterations bounded by the length of the longest path in the body of L. The result follows immediately. □

Theorem 9.4 (Convergence) For a sufficiently large unwinding i, on every path in $\mathcal{C}u^iL$ there exists a node n such that there is another node n' (not necessarily on the same path) equivalent to n.

let k = length of longest path in the loop body L;
$P \leftarrow Cu^k L$;
for each path p through P **do**
  let n be the first node on p s.t. no operation
    in n is from iteration L_1.
   Find n' equivalent to n;
   Replace edges (m, n') by (m, n);
   Delete n' and any other unreachable nodes;

Figure 14: Perfect Pipelining.

Proof: Property 2 assures the existence of n and n', as every node can have operations from some fixed range of iterations and there are no more than c operations per node, implying that there are only finitely many distinct classes of equivalent nodes. □

This theorem combined with Property 1 shows that a loop repeatedly unwound and compacted using C eventually falls into a repeating pattern. The pattern itself may be very complex, but it is sufficient to find two equivalent nodes to detect when it repeats. For the simple rule, if k is the length of the longest path in the loop body, then unwinding k times guarantees convergence on every path.

The Perfect Pipelining transformation is given in Figure 14. The algorithm finds equivalent nodes n and n' in the compacted program graph, deletes n', and adds backedges from the predecessors of n' to n. For the simple rule, it can be shown that the first node on any path without an operation from the first iteration is repeated.

Lemma 9.5 Let $Cu^\infty L$ be the result of the application of Perfect Pipelining. For any integer k, there is an unwinding i such that $T(Cu^\infty L, s)$ is identical to $T(Cu^i L, s)$ for the first k steps.

Proof: Let $i \gg k$. The proof is by induction.

Basis: The pair $\langle n_0, s\rangle$ is the initial configuration of both $T(Cu^\infty L, s)$ and $T(Cu^i L, s)$. As iterations do not completely overlap, the first node in both programs contains the first operation of the first iteration.

Step: Assume that $\langle n_0, s\rangle \xrightarrow{*} \langle n_j, s_j\rangle$ in both programs. Let $\langle n_j, s_j\rangle \rightarrow \langle n_{j+1}, s_{j+1}\rangle$ in $Cu^i L$, and let $\langle n_j, s_j\rangle \rightarrow \langle n, s_{j+1}\rangle$ in $Cu^\infty L$. In the first k steps $Cu^i L$ does not complete the execution of even a single iteration, as $Cu^i L$ has at least i nodes on the shortest path in its loop body. Because $i \gg k$, further unwinding and compaction using $\mathcal{C}$ cannot increase the number of operations in n_{j+1}. If n is not on one of the backedges introduced by the algorithm, then by Property 1 and the unwinding chosen in the algorithm $n = n_{j+1}$. If the successor is across a backedge, then let n' be the node that was replaced by the algorithm. Again, by Property 1 and the choice of unwinding, $n' = n_{j+1}$. But then $n = n'$ and the lemma follows. □

Theorem 9.6 For all i and L satisfying the dependency constraint, $\mathcal{C}u^iL \leq_p \mathcal{C}u^\infty L$.

Proof: Let k be the length of $T(\mathcal{C}u^iL, s)$. Consider a program $\mathcal{C}u^jL$ where $j \gg \mathbf{max}(i, k)$. By the previous lemma, $T(\mathcal{C}u^jL, s) = T(\mathcal{C}u^\infty L, s)$. Because $\mathcal{C}$ is well-behaved, $\mathcal{C}u^iL \leq_p \mathcal{C}u^jL$. We conclude that $\mathcal{C}u^iL \leq_p \mathcal{C}u^\infty L$. □

This theorem shows that Perfect Pipelining is as good as full unwinding and compaction on all paths. The transformation computes a closed form of the pattern generated by repeated unwinding and compaction using $\mathcal{C}$.

9.1 An Example

Refer again to the loop in Figure 1a. The result of applying Perfect Pipelining to this loop is shown in Figure 1d. The length of the loop body of the original loop is four; in Figure 1c the loop has been unwound five times and compacted using $\mathcal{C}$. The fourth and fifth nodes are equivalent. The transformation deletes the fifth node and all succeeding nodes and adds an edge from the fourth node to itself with an induction variable increment of one (the increment is the number k in Definition 9.1).

10 Pipelining with Limited Resources

Thus far we have assumed that our abstract machine has unlimited resources. In practice, compilers must consider the fact that parallel computers have restrictions on the number of operations of a particular type that can be executed simultaneously. In our program graph representation, a node may not contain more than a fixed number of operations of a given type.

The modification to Perfect Pipelining is made in the Move transformation (Figure 11). The change is simple: an operation may not move into a node if the node then violates the resource constraints.

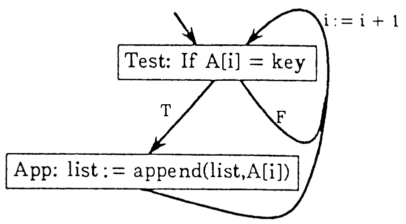


Figure 15: A simple loop L.

Lemma 10.1 Properties 1 and 2 (Lemmas 9.2 and 9.3) hold in the presence of resource constraints.

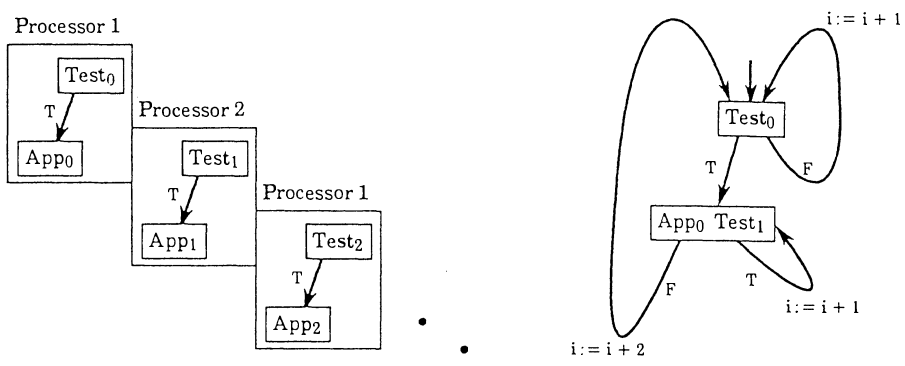
Proof: Property 2 clearly holds. Consider Property 1. Let p and p' be corresponding successors of n and n' as in Lemma 9.2. The only change to the original proof is to note that if the first operation of an iteration L_j is in p, then the resources for this operation must also be available in p'. □

The proofs of Theorems 9.4 and 9.6 apply immediately when Properties 1 and 2 hold. Resource constraints guarantee Property 2 (Lemma 9.3) by imposing a fixed upper bound on the size of program nodes. Thus, the simple rule applies to all loops satisfying Constraint 7.1 without the stronger condition used in Section 9.

Figure 15 shows a simple loop L. The loop searches an array of elements, saving the position of all elements that match a key in order on a separate list. As before, we have left the details of the loop control code implicit. There is also no exit test; we stress that this is only for simplicity. We assume that the target machine can execute up to three tests in parallel.

This particular loop highlights the problem that unpredictable flow of control presents in parallelization. Note that while the path corresponding to the true branches has tight dependencies preventing speedup, the path corresponding to the false branches has no dependencies whatsoever. Other paths (some true branches, some false branches) have intermediate parallelism.

Existing restructuring transformations for multiprocessors can do very little with such a loop. Doacross is a transformation that assigns the iterations of a loop to the processors of a synchronous or asynchronous multiprocessor [Cyt86]. Doacross computes a delay that must be observed between the start of a loop iteration L_i and the start of L_{i+1} on each path of L_i. For this loop, the computed delay is one on both paths; i.e., iteration $i+1$ may begin after iteration i has executed its first statement. The dynamic execution of this



(a) Dynamic schedule. (b) Static SPL program.

Figure 16: Doacross applied to L.

loop using Doacross is shown in Figure 16a. An equivalent static SPL schedule is shown in Figure 16b.

We now show how Perfect Pipelining applies to this loop. Figure 17 shows the original loop unwound six times. The operations have been replaced by labels with subscripts indicating the increment to the induction variable. The result of applying the simple rule is shown in Figure 18. The dag of tests within each node is arranged as a chain with the false branches pointing to the next test and the true branches exiting the node; the lowest numbered test is the root of the dag.

The first four nodes in the left column of Figure 18 are equivalent and the start node is equivalent to the first two nodes in the right column. Figure 19 shows the result of applying Perfect Pipelining—only the first two nodes remain. In this program, three tests are performed in parallel. If T_j is the lowest numbered test that evaluates to true, then the induction variable i is incremented by j and control passes to the node with the append operation. If none of the tests is true, control transfers to the first node. The second node performs an append and evaluates the next three tests.

The pipelined loop executes three tests at every step, achieving optimal use of the critical resource. The number of items matching the key can be relatively large without dramatically affecting the speedup. The final code can run on the Multiflow machine, a commercial tightly-coupled parallel architecture that supports multiway jumps. The running time of Perfect Pipelining with resource constraints is dependent on the size (number of resources) of the machine as well as the original loop.

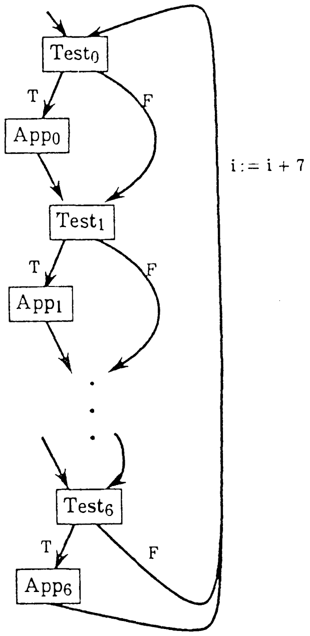


Figure 17: L unwound six times.

11 Comparison with Doacross

As suggested in the previous section, loops transformed by Doacross can be represented in our formalism. In fact, a restriction on the pipelining transformation corresponds exactly to Doacross for single loops on synchronous multiprocessors. Another, more restrictive version corresponds to Doacross for asynchronous multiprocessors. Thus a family of transformations aimed at different machine models can be directly formulated and compared in our framework.

The basic algorithm for Doacross analyzes a loop body and decides where, on each path, it is safe to begin the next iteration. A communication instruction is added to the loop at those points. During execution, when a processor executing iteration i encounters a communication instruction, it sends a message signaling another processor that execution of iteration $i + 1$ can begin.

Let $\mathcal{D}_{synch}$ be the compaction operator implementing Doacross for synchronous multiprocessors. The restriction to the pipelining algorithm is made in Move (see Figure 20). The new requirement is that if an iteration moves above a test, then it must move above that test on all paths. This restriction is necessary for Doacross because the various processors have independent flow of control—once an iteration is started on a processor it must be able to proceed regardless of the path taken by any other processor. It is easily shown

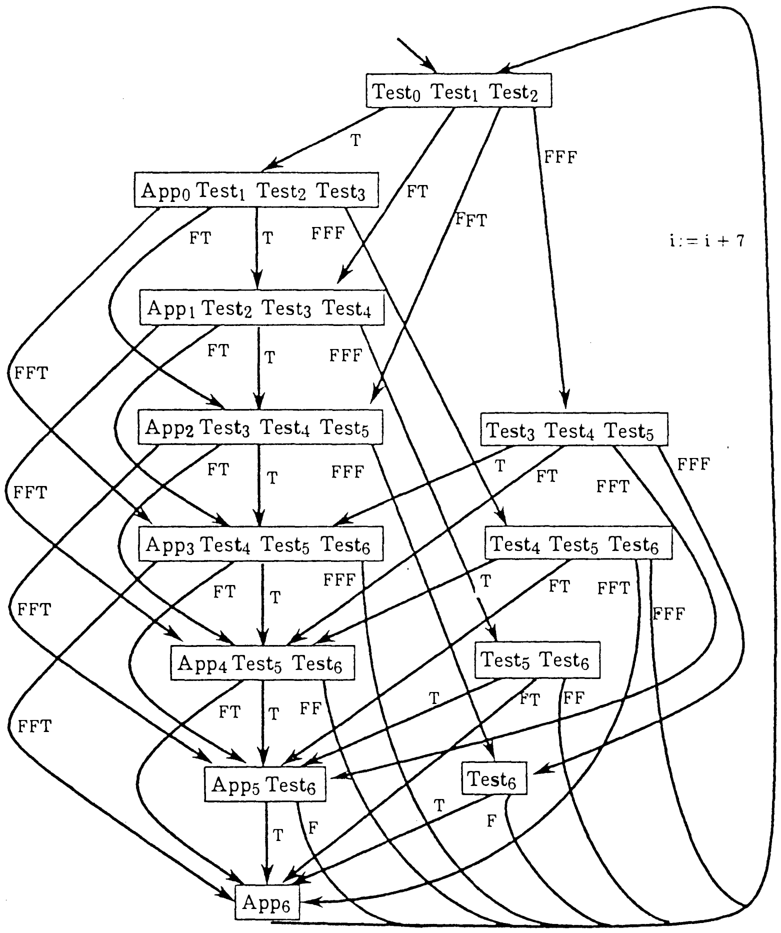


Figure 18: L unwound six times and compacted.

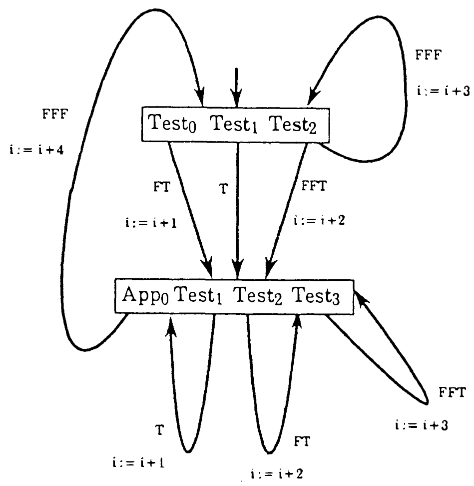


Figure 19: The same loop after pipelining.

move(x, n, m)
    if x is an assignment
        then if $x \in op(s)$ for all $s \in succ(n)$
            $P \leftarrow unify(P, x, n, m)$
        else $P \leftarrow move\text{-}test(P, x, n, m)$
    if no change in P
        then return (*False*)
        else return (*True*)

Figure 20: The Move operator for $\mathcal{D}_{synch}$.

that for $\mathcal{D}_{synch}L$, the first operation of iteration $i+1$ overlaps iteration i exactly where the communications are introduced by Doacross. The asynchronous case ($\mathcal{D}_{asynch}$) is similar and can also be written as a restriction on the pipelining transformation; however, the communication points must be introduced even more conservatively because the processors do not run in lock-step. The following theorem precisely summarizes the relationship between the three transformations.

Theorem 11.1 For all loops L, $\mathcal{D}_{asynch}L \leq_p \mathcal{D}_{synch}L \leq_p \mathcal{C}u^{\infty}L$.

Proof: Follows immediately from previous discussion and the fact that $\mathcal{D}_{asynch}$ is a restriction of $\mathcal{D}_{synch}$, which itself is a restriction of Perfect Pipelining. In cases where loop carried dependencies are not regular, conservative assumptions about dependencies can be made for Perfect Pipelining and the theorem holds. The theorem applies for both bounded and unbounded resources (see the appendix). □

12 Conclusion

We have presented a new technique, Perfect Pipelining, that allows full fine-grain parallelization of loops. Perfect Pipelining is currently being integrated into ESP, an Environment for Scientific Programming under development at Cornell. The environment already includes Percolation Scheduling and several other transformations. We believe that Perfect Pipelining will greatly enhance the power of our environment by subsuming the effects of a class of coarser-grain transformations in a uniform, integrated fashion compatible with our fine-grain approach.

13 Acknowledgements

The first author would like to thank Anne Neirynck for many hours of discussion on the problems of providing formalisms for parallelizing transformations. Prakash Panangaden provided a great deal of helpful advice on almost all aspects of this work. Laurie Hendren, Prakash Panangaden, and Jennifer Widom criticized drafts of this paper and contributed greatly to its final form.

A Appendix

A.1 Efficiency

There are loops satisfying Constraint 7.1 for which Perfect Pipelining requires exponential time. In particular, if there are no loop carried dependencies at all—iterations are completely independent—then the running time is exponential in the unwinding if there is at least one test in the loop body. However, this can be detected after unrolling only once, because the iterations completely overlap after applying $\mathcal{C}$. In this case, the loop is completely vectorizable and generating good code is relatively easy.

It is also possible to construct examples with some loop carried dependencies for which Perfect Pipelining requires exponential time. However, several conditions need to be simultaneously satisfied for this to happen. We believe that these conditions do not commonly arise in practice. In fact, for every program we have examined (including the examples in this paper and all of the Livermore Loops) the pipelining algorithm runs in low-order polynomial time and requires at most quadratic space. Convergence often occurs on many or all paths for unrollings much smaller than the worst case bound; thus interleaving unwinding, compaction, and the test for equivalent nodes substantially improves the efficiency of the algorithm. Using simple data structures, the check for equivalent nodes can be done very quickly.

A.2 Bounded Operators

The simple rule is only one of a family of compaction operators that work in the pipelining transformation. The following theorem generalizes the result.

Theorem A.1 Consider $\mathcal{C}'u^iL$, where $\mathcal{C}'$ is a well-behaved compaction operator and L satisfies Constraint 7.1. If there is a constant d, independent of the unrolling chosen, such that on any path in the compacted loop body the first and last scheduled operations of L_j are no more than d nodes apart, then Theorem 9.4 holds.

The proof of the theorem is similar to the original proof of convergence. The idea is that because all iterations are confined to a region of size d, there are only finitely many possible arrangements of operations in an iteration and only finitely many iterations may overlap; for a large enough unwinding a node must repeat.

Unfortunately, only a weak version of Property 1 holds. Because the order of operations in an iteration is no longer fixed, some pairs of equivalent nodes may not have equivalent successors. While the convergence test can still be done automatically, it is more expensive than simply detecting equivalent nodes. In addition, the rate of convergence—amount of unwinding required—increases linearly with the number of distinct iteration shapes generated by C'. Because there can be exponentially many shapes (distinct arrangements of operations in a loop body), any practical variation on the pipelining transformation cannot deviate too much from the simple rule.

A.3 Dependency Constraints

In Section 9 we required that there be loop carried dependencies on all paths of a loop. If such dependencies do not exist on a path, then the simple rule might completely overlap iterations on that path—i.e., a copy of every unwound iteration rises to the top of the loop on that path. Without resource constraints, nodes grow larger without bound with additional unwinding and application of the simple rule, so Property 2 does not hold and Perfect Pipelining cannot be applied.

This problem does not arise in the bounded resource case. A simple change to Perfect Pipelining overcomes this problem in the unbounded case. We add a rule that the first operation of an iteration may not move into a node containing the first operation of another iteration. This rule "pretends" that there is a loop carried dependency between the first operations of consecutive iterations. If there are loop carried dependencies on a given path then the change has no effect; otherwise, it permits the application Perfect Pipelining to a loop satisfying Constraint 7.1. This justifies the claim that Theorem 11.1 applies to both bounded and unbounded resources.

A.4 Formal Definitions of the Core Transformations

Completely formal proofs of the core transformations require tedious (and relatively uninformative) inductions on the length of the trace of an execution. We present informal proofs that highlight the important properties of the transformations. Minor variations on these proofs can be used to show that $P \leq_p \mathcal{T}(P)$ for any of the primitive transformations $\mathcal{T}$.

Figure 21 lists predicates and functions used to define the core transformations. We briefly describe each here. The function *attach* builds a new dag out of two dags and a test x. The second and third arguments correspond to the true and false branches of x respectively; x is the root of the resulting dag. *Reach* returns the subdag reachable (in the sense of a directed graph) from a test or name t. The predicate *empty* is true if its argument is a node with no operations. A node is *unreachable* if it has no predecessors and is not the start node. The function *replace_edge* returns a dag with names pointing to node n replaced by names pointing to n'.

The predicates *dead*, *no_write_conflict*, and *no_read_conflict* are used to determine when an operation may safely move from one node to another. These predicates model the flow analysis and disambiguation mechanisms of parallelizing compilers. *No_write_conflict*(x, n) is true if the operation x does not read or write a value written by an operation in n. *No_read_conflict*(x, n) is true if x does not write a value read by an operation in n. *Dead(l,n)* is true if location l does not affect further evaluation of the program when control reaches n.

A.4.1 Delete

The formal definition of Delete is given in Figure 22. A picture is given in Figure 6. Note that an empty node has exactly one successor.

Lemma A.2 Delete is correct.

Proof: Let $delete(P, n) = P'$. Consider traces $T(P, s)$ and $T(P', s)$. Because delete preserves flow of control and does not remove any operations from P, it is clear that trace dependency graphs of the traces are identical. Thus, P is strongly equivalent to P'. □

A.4.2 Move-op

The formal definition of Move-op is given in Figure 23. A picture is given in Figure 7.

Lemma A.3 Move-op is correct.

Proof: Let $P' = move\text{-}op(P, x, n, m)$. Clearly, if P' does not enter the region of the program modified by the transformation, then P' behaves exactly as P. Thus the traces $T(P, s)$ and $T(P', s)$ are identical up to the step that P' first executes either n or m. There are three cases:

1. Both programs execute n. In this case the two programs execute the same node and execution continues outside the affected region.

attach: Test $\times$ Dag $\times$ Dag $\rightarrow$ Dag

Let $C_1 = \langle B_1, sel_1, r_1, H_1\rangle$, $C_2 = \langle B_2, sel_2, r_2, H_2\rangle$

$attach(t, C_1, C_2) = \langle B_1 \cup B_2 \cup \{t\}, sel[\langle t, \mathrm{tt}\rangle \leftarrow r_1, \langle t, \mathrm{ff}\rangle \leftarrow r_2], t, H_1 \cup H_2\rangle$

where $sel = sel_1 \sqcup sel_2$

reach: (Dag $\times$ (Test + Name)) $\rightarrow$ Dag

Let $C = \langle B, sel, r, H\rangle$

$$reach(C, t) = \begin{cases} \perp & \text{if } t \notin B, H \\ \langle \emptyset, \lambda x.\ \perp, t, \{t\}\rangle & \text{if } t \in H \\ attach(t, reach(C, sel(t, \mathrm{tt})), reach(C, sel(t, \mathrm{ff}))) & \text{otherwise} \end{cases}$$

empty: Node $\rightarrow$ Bool

$empty(n) \equiv A = \emptyset \wedge B = \emptyset)$

unreachable: Node $\rightarrow$ Bool

$unreachable(n) \equiv (pred(n) = \emptyset \wedge n \neq n_0)$

replace_edge: Dag $\times$ *list_of*(Name) $\rightarrow$ Dag

$replace\_edge(\langle B, select, r, H\rangle, n_1, n_1', \ldots, n_i, n_i') = \langle B, select', r', H'\rangle$

where $H' = H\left[n_j'/n_j\right]$

$select' = select\left[\langle z, b\rangle \leftarrow n_j'\right]$

where $select(z, b) = n_j$

$$r' = \begin{cases} n_j' & \text{if } r = n_j \\ r & \text{otherwise} \end{cases}$$

no_write_conflict: (Test + Assign) $\times$ Node $\rightarrow$ Bool

$no\_write\_conflict(x, n) \equiv \forall a, s \in A\ (read(x, s) \cup write(x, s)) \cap write(a, s) = \emptyset$

no_read_conflict: (Test + Assign) $\times$ Node $\rightarrow$ Bool

$no\_read\_conflict(x, n) \equiv \forall a, s \in A\ write(x, s) \cap read(a, s) = \emptyset$

where $write(x, s) = \emptyset$ if x is a test, $n = \langle A, C\rangle$

dead: Loc $\times$ Node $\rightarrow$ Bool

$\forall s$: Store, v : Val, $\langle n_0, s\rangle \xrightarrow{*} \langle n_i, s_i\rangle \xrightarrow{*} \langle n_{i+1}, s_{i+1}\rangle \xrightarrow{*} \langle n_k, s_k\rangle \Rightarrow \langle n_i, s_i[l \leftarrow v]\rangle \xrightarrow{*} \langle n_{i+1}, s_{i+1}'\rangle \xrightarrow{*} \langle n_k, s_k\rangle$

Figure 21: Some useful definitions.

delete: Pgm $\times$ Node $\rightarrow$ Pgm

Let $P = \langle N, n_0, F \rangle$

$n = \langle A, C \rangle$

$C = \langle B, select, r, H \rangle$

$p \in pred(n),\ p = \langle A_p, C_p \rangle$

$delete(P, n) \equiv$ **if** $empty(n) \vee unreachable(n)$ **then** P' **else** P

where:

$P' = \langle N', n_0', F \rangle$

$N' = N - \{n\}$

if $\neg unreachable(n)$**then**

let $succ(n) = \{n'\}$

$$n_0' = \begin{cases} n' & \text{if } n = n_0 \\ n_0 & \text{otherwise} \end{cases}$$

$p = \langle A_p, C_p' \rangle$ where $C_p' = replace\_edge(C_p, n, n')$

Figure 22: Definition of delete.

move-op: Pgm $\times$ Assign $\times$ Node $\times$ Node $\rightarrow$ Pgm

Let $P = \langle N, n_0, F \rangle$

$m = \langle A_m, C_m \rangle$

$n = \langle A_n, C_n \rangle$

$move\text{-}op(P, x, n, m) \equiv$ **if** $Cond$ **then** P' **else** P

where:

$Cond \equiv n \notin F \wedge no\_read\_conflict(x, n) \wedge no\_write\_conflict(x, m) \wedge \forall p, l\ dead(l, p)$

where $p \in (succ(m) - \{n\})$ and $l \in write(x)$

$P' = \langle N', n_0, F \rangle$

$N' = N \cup \{n'\}$

$m = \langle A_m', C_m' \rangle$ where $A_m' = A_m \cup \{x\}$, $C_m' = replace\_edge(C_m, n, n')$

$n' = \langle A_n', C_n \rangle$ where $A_n' = A_n - \{x\}$

Figure 23: Definition of move-op.

unify: Pgm $\times$ Assign $\times$ Node $\times$ Node $\rightarrow$ Pgm

Let $P = \langle N, n_0, F \rangle$

$m = \langle A_m, C_m \rangle$

$move\_set = \{n_j\}$ s.t. $n_j \in succ(m) \wedge x \in A_j$

where $n_j = \langle A_j, C_j \rangle$

$unify(P, x, n, m) \equiv$ **if** $Cond$ **then** P' **else** P where:

$Cond \equiv n \notin F \wedge \forall n \in move\_set\ no\_read\_conflict(x, n) \wedge$

$no\_write\_conflict(x, m) \wedge \forall p, l\ dead(l, p)$

where $p \in (succ(m) - move\_set)$ and $l \in write(x)$

$P' = \langle N', n_0, F \rangle$

$N' = N \cup \{n'_j | n_j \in move\_set\}$

$m = \langle A'_m, C'_m \rangle$ where $A'_m = A_m \cup \{x\}$, $C'_m = replace\_edge(C_m, \ldots, n_j, n'_j, \ldots)$

$n'_j = \langle A'_j, C_j \rangle$ where $A'_j = A_j - \{x\}$

Figure 24: Definition of unify.

2. P' executes m and then n'; P executes m and then n. It is easily verified that, because of the condition placed on the transformation, the effect of the assignments in P and P' is identical.

3. P' and P execute m and the next node is not n. In this case whatever x writes in P' is dead—see the transformation condition—and can have no affect on the computation.

Proceeding by induction on the number of times P' enters the transformed part of the program, we see that in a final state the final stores of the two programs can only differ in locations that are dead. As the final operation reads the entire store (i.e., no locations are dead) in both traces, we conclude that $R(P, s) = s' \Leftrightarrow R(P', s) = s'$.[5] □

A.4.3 Unify

The formal definition of Unify is given in Figure 24. A picture is given in Figure 8.

Lemma A.4 Unify is correct.

Proof: The proof is very similar to that for Move-op and is omitted. □

[5]It is not necessary for correctness that the entire final store be the result of of a program; this approach has been adopted for simplicity.

move-test: Pgm $\times$ Test $\times$ Node $\times$ Node $\to$ Pgm

Let $P = \langle N, n_0, F \rangle$

$m = \langle A_m, C_m \rangle$

$C_m = \langle B_m, select_m, r_m, H_m \rangle$

$n = \langle A_n, C_n \rangle$

$C_n = \langle B_n, select_n, r_n, H_n \rangle$

$move\text{-}test(P, x, n, m) \equiv$ **if** $no\_write\_conflict(x, m)$ **then** P' **else** P

where:

$P' = \langle N', n_0, F \rangle$

$N' = N \cup \{n_t, n_f\}$

$m = \langle A_m, C'_m \rangle$ where:

$C'_m = \langle B'_m, select'_m, r_m, H'_m \rangle$

$H'_m = H_m - \{n\} \cup \{n_t, n_f\}$

$B'_m = B'_m \cup \{x\}$

$select'_m = select_m\left[\langle x, \mathrm{tt} \rangle \leftarrow n_t, \langle x, \mathrm{ff} \rangle \leftarrow n_f, \langle z, \mathrm{b} \rangle \leftarrow x\right]$

where $select_m(z, b) = n$

$n_t = \langle A_n, C'_n \rangle$ where:

$C'_n = reach(\langle B_n, select_n\left[\langle z, \mathrm{b} \rangle \leftarrow select_n(x, \mathrm{tt})\right], r_n, H_n \rangle, r_n)$

where $select_n(z, b) = x$

$n_f = \langle A_n, C''_n \rangle$ where:

$C''_n = reach(\langle B_n, select_n\left[\langle z, \mathrm{b} \rangle \leftarrow select_n(x, \mathrm{ff})\right], r_n, H_n \rangle, r_n)$

where $select_n(z, \mathrm{b}) = x$

Figure 25: Definition of move-test.

A.4.4 Move-test

The formal definition of Move-test is in Figure 25. A picture is given in Figure 9.

Lemma A.5 Move-test is correct.

Proof: Let $P' = \textit{move-test}(P, x, n, m)$. As with Move-op, if P' behaves exactly as P up to the step that it first enters the region affected by the transformation. At that point there are four cases:

1. Both execute n. In this case the two programs execute the same node and execution continues outside the affected region.

2. P' and P execute m and the next node is not n. Test x in m is not on any paths through m's dag of tests except the paths to n_t and n_f. Thus, the evaluation of x has no effect in this case.

3. P' executes m and then n_t; P executes m and then n. The nodes n_t and n differ at one point in the dag of tests: n has x at that point while n_t has the subdag rooted at x's true branch in n. Thus, if x evaluates to true in n, then the next node must be the same in each trace. By the condition of the transformation, because x evaluates to true at m in P', x evaluates to true at n in P.

4. P' executes m and then n_f; P executes m and then n. This is symmetric to the previous case.

Proceeding by induction on the number of times P' enters the transformed part of the program, we see that the traces execute the same assignments at each step, and therefore the stores are always equal. Thus the final stores must be equal and the programs are strongly equivalent. □

A.4.5 Unrolling

The formal definition of unrolling is given in Figure 26. A picture is given in Figure 10. In the definition, the superscript of an operation is the unrolled iteration to which the operation belongs, while the subscript distinguishes operations within a loop body. The use of *replace_edge* ensures that a node's successors are from the same unrolled iteration or, if a backedge is being replaced, that the node points to the first node of the next unrolled iteration. The correctness of unrolling follows immediately from the following lemma.

Unroll: Pgm $\times$ $\mathcal{P}$(Node) $\rightarrow$ Pgm

Let $P = \langle N, n_0, F \rangle$

$M = \{m_1, \ldots m_k\}$ be a set of nodes, satisfying:

M is a maximal connected region of P

m_1 is the unique entry point to M

$m_1 \in succ(m_k)$ (m_k, m_1) is the unique backedge of M

$unroll(P, M, i) = P'$ where:

$P' = \langle N', n_0, F \rangle$

$N' = N \bigcup_{1 \leq a \leq i, 1 \leq b \leq k} \{m_b^a\}$

Let $m_b^a = \langle A_b, C_b^a \rangle$ where:

$\forall a, b$ s.t. $1 \leq a \leq i \wedge 1 \leq b < k$ $C_b^a = replace\_edge(C_b, m_1, m_1^a, m_2, m_2^a, \ldots)$

$\forall a$ s.t. $1 \leq a < i$ $C_k^a = replace\_edge(C_b, m_1, m_1^{a+1}, m_2, m_2^a, m_3, m_3^a, \ldots)$

$C_k^i = replace\_edge(C_b, m_2, m_2^i, m_3, m_3^i, \ldots)$

Figure 26: Definition of unrolling.

Lemma A.6 Let $P' = unroll(P, \{m_h\}, i)$. Then $\langle n_0, s_0 \rangle \xrightarrow{*} \langle n_j, s_j \rangle$ in P implies that $\langle n'_0, s_0 \rangle \xrightarrow{*} \langle n'_j, s_j \rangle$ in P' where:

$$n_j = n'_j \vee (n_j = m_h \wedge n'_j = m_h^g)$$

Proof: The proof is by induction on the length of $T(P, s)$ and $T(P', s)$.

Basis: Trivial.

Step: Assume the hypothesis for $\langle n_0, s_0 \rangle \xrightarrow{*} \langle n_j, s_j \rangle$ and $\langle n'_0, s_0 \rangle \xrightarrow{*} \langle n'_j, s_j \rangle$. There are two cases:

1. $n_j = n'_j$. This case is trivial as the nodes are identical and evaluated in the same store.

2. $n_j = m_h \wedge n'_j = m_h^g$. The nodes contain identical operations; only the names in the dags are different. It is easily checked that the possible outcomes satisfy the requirements of the induction: either a name was not replaced, in which case $n_{j+1} = n'_{j+1}$ or it was replaced, in which case $n_{j+1} = m_{h'} \wedge n'_i = m_{h'}^{g'}$.

□

References

[AK84] J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, pages 233–246, June 1984.

[AN87] A. Aiken and A. Nicolau. *Loop Quantization: an Analysis and Algorithm.* Technical Report 87-821, Cornell University, 1987.

[AN88] A. Aiken and A. Nicolau. A development environment for horizontal microcode. *IEEE Transactions on Software Engineering*, To Appear 1988. Also available as Cornell Technical Report TR 86-785.

[CCK87] D. Callahan, J. Cocke, and K. Kennedy. *Estimating Interlock and Improving Balance for Pipelined Architectures.* Technical Report COMP TR-87-51, Rice University, May 1987.

[Cyt86] R. Cytron. Doacross: beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836–844, August 1986.

[FERN84] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: a smart compiler and a dumb machine. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, pages 37–47, June 1984.

[Fis81] J. A. Fisher. Trace Scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–90, July 1981.

[KM75] D. J. Kuck and Y. Maruyama. Time bounds on the parallel evaluation of arithmetic expressions. *SIAM Journal of Computing*, 4:147–62, June 1975.

[KN85] K. Karplus and A. Nicolau. Efficient hardware for multi-way jumps and prefetches. In *Proceedings of the 18th Annual Workshop on Microprogramming*, pages 11–18, December 1985.

[Kuc76] D. Kuck. Parallel processing of ordinary programs. In *Advances in Computers*, pages 119–179, Academic Press, New York, 1976.

[Lam74] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[MS87] A. Munshi and B. Simons. *Scheduling Sequential Loops on Parallel Processors.* Technical Report 5546, IBM, 1987.

[Nic85a] A. Nicolau. *Loop Quantization, or Unwinding Done Right.* Technical Report 85-709, Cornell University, 1985.

[Nic85b] A. Nicolau. Uniform parallelism exploitation in ordinary programs. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 614–618, August 1985.

[Plo] G. D. Plotkin. A structural approach to operational semantics. Text prepared at University of Aarhus.