Perfect Pipelining: A New Loop Parallelization Technique*

Alexander Aiken
Alexandru Nicolau
Computer Science Department
Cornell University
Ithaca, New York 14853 USA

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. Coarse 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. 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 on 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 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.

The program graph in Figure 1a illustrates the importance of Perfect Pipelining. (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 three times and compacted. (Two additional memory locations are allocated to each array to handle the extra references generated when $i = n$.) 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 five 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 this becomes expensive very rapidly. Existing compaction transformations can achieve the schedules in Figures 1b

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

(a) Original loop.

(b) Unwound three times and compacted.

(c) Unwound five times and compacted.

(d) Loop after pipelining.

Figure 1: A Perfect Pipelining example.

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 creates 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. 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; $\leq_p$ 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 resulting formalism 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 *Doacross* [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. 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

Bool	=	tt + ff
Loc	=	$\mathcal{Z}$
Store	=	Loc $\rightarrow$ Val
Assign	=	Store $\rightarrow$ Loc $\times$ Val
Test	=	Store $\rightarrow$ Bool

(a) Basic domains.

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 $x \in op(n)$

(b) Useful functions.

Figure 2: Some definitions.

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.

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 2a 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 $\langle B, select\ , r, H\rangle$ where:
  B is a set of *tests*
  $select\ : B \times \text{Bool} \rightarrow B + H$ 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 clear from the context.

The transformations we define require knowledge of the locations that are read and 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. We omit the formal definition of these requirements; the details can be found in [AN87b]. 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.

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

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

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

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

Figure 3: Operational semantics of SPL.

Figure 3 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: $\rightsquigarrow$, which defines transitions within a node, and $\rightarrow$, 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 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 $\forall s\ R(P, s) = s' \Leftrightarrow R(P', s) = s'$.

If $\mathcal{T}$ is a program transformation, then $\mathcal{T}$ is *correct* if $\mathcal{T}(P)$ is strongly equivalent to P for all P. We require that transformations be correct; this guarantees that any sequence of transformations is strongly equivalent to the original program. The formal definitions of the transformations and proofs of correctness can be found in [AN87b]. In this paper, we briefly describe and illustrate each transformation.

Figure 2b 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 $n \in \mathit{succ}\ (m)$. *Pred* returns the immediate predecessors of a node. The function *op* returns the operations in a node. $\mathit{Node}(x)$ is the node containing operation x (we assume there is some way of distinguishing between multiple copies of the same operation).

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 4a 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.

The *Unify* transformation moves a single copy x of identical assignments from a set of nodes $\{n_j\}$ to a common predecessor node m. This is done if no dependency exists between x and the operations of m and x does not kill any value live at m. Care must be taken not to affect the computation of paths passing through n but not through m. To ensure this, the original node n is preserved on all other paths. An illustration is given in Figure 4b.

(a) The delete transformation. (b) The unify transformation.

Figure 4: Primitive transformations.

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 5. 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.

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, AN87a, CCK87]. The shorthand $u^i L$ denotes the loop where i copies of the loop body of L are unrolled.

5 A Formalization of Parallelism

In this section we develop a formal account of our transformations. This allows 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 relation $\leq_p$ 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. The purpose of **sim** is to establish a dependency-preserving mapping between operations in traces of P and operations in traces of P'.

Definition 5.1 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 $write(x, s_i) = write(z, s_k)$.

Figure 5: The move-test transformation.

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 relation $\prec$ precisely captures the flow of values through an execution of a program. This is all that is required to define the relation **sim**.

Definition 5.2 (Similarity) P **sim** P' if and only if there exists a function f satisfying:

$$\begin{aligned} \forall s \quad & x \prec y \text{ in } T(P,s) \Rightarrow f(x) \prec f(y) \text{ in } T(P',s) \wedge \\ & f(x) \prec y' \text{ in } T(P',s) \Rightarrow x \prec f^{-1}(y') \text{ in } T(P,s) \end{aligned}$$

where f is 1-to-1 from operations in $T(P,s)$ to operations in $T(P',s)$ and $f(x)$ is an occurrence of x.

The function f provides 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. We use $\leq_p$ to prove that some improvement results from the application of the core transformations.

Definition 5.3 Let $x \in op\ (n_i)$ in $T(P,s)$. The position of x, written $pos\ (x)$, is i.

$$P \leq_p P' \Leftrightarrow P \textbf{ sim } P' \wedge \forall s\ pos\ (x) \text{ in } T(P,s) \geq pos\ (f(x)) \text{ in } T(P',s)$$

Theorem 5.4 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 $P \leq_p \mathcal{T}(P)$. □

6 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. 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 rapidly and there will still be a "tail" of sequential code at the 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.

6.1 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 [AN87a].

Consider the sequence $\mathcal{C}uL, \mathcal{C}u^2L, \mathcal{C}u^3L, \ldots$. If $\forall i\ \mathcal{C}u^iL \leq_p \mathcal{C}u^{i+1}L$, then $\mathcal{C}$ is *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

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

6.2 The Programs

A loop u^iL consists of unwound iterations $L_1, \ldots, L_i$. 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 simple loops satisfying the following property for any unwinding:

Constraint 6.1 Assume there is a loop carried dependency between operations x and y in L. Then in u^iL, there is a dependency between operations 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.

7 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 [AN87b]. Initially we assume that computational resources are unlimited; in Section 9 we discuss Perfect Pipelining when resources are bounded.

7.1 The Simple Rule

To simplify the algorithms, we combine the primitives Unify and Move-test into one operation *Move* (see Figure 6a). 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.

One step of the simple rule moves each operation in one copy of an iteration up one node in the program graph. An algorithm that accomplishes this is given in Figure 6b. 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.

The simple rule is given in Figure 7. 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, $\mathcal{C}u^iP \not\leq_p \mathcal{C}'u^iP$. 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.

8 Perfect Pipelining

In this section, we require that loop carried dependencies satisfy Constraint 6.1 and that there be enough such dependencies that $\mathcal{C}$ cannot completely overlap unwound iterations on any path. In Section 9 we remove this stronger condition. The following two properties of the simple rule are required for Perfect Pipelining. Proofs of lemmas not included in this paper may be found in [AN87b].

Definition 8.1 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 8.2 (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.

Lemma 8.3 (Property 2) There is a constant c, dependent only on L, satisfying

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

Theorem 8.4 (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.

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 $\mathcal{C}$ 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, it is sufficient to unwind $k+1$ copies of L to find the pattern on every path, where k is the length of the longest path in the loop body. The Perfect Pipelining transformation is given in Figure 8. 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 8.5 Let $\mathcal{C}u^\infty L$ be the result of the application of Perfect Pipelining. For sufficiently large unwindings i, $T(\mathcal{C}u^\infty L, s)$ is identical to $T(\mathcal{C}u^iL, s)$ for the first $i/2$ steps.

Theorem 8.6 For all i and L satisfying the dependency constraint, $\mathcal{C}u^iL \leq_p \mathcal{C}u^\infty L$.

```
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)
```

(a) 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;
```

(b) Moving an iteration.

Figure 6: Higher-level transformations.

```
(* Let P = u^i L *)
for each iteration L_1, ..., L_i do
    X ← {x} where x is the first operation in L_j
    repeat
        (* we assume that X always contains all copies of operation x *)
        1. while ∃y ∈ X s.t. pred(node(y)) = {p} and y's iteration can move
            do move_iteration (y, node(y), p)

        2. if ∃y ∈ X s.t. y can move to node p ∈ pred(node(y))
            and the rest of the iteration can move accordingly then
                select y s.t. the depth of node(y) in the program graph is maximized;
                move_iteration (y, node(y), p)
    until 2 fails.
    Delete all empty nodes.
```

Figure 7: The simple rule.

```
let k = length of longest path in the loop body L;
P ← Cu^{k+1} 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 8: Perfect Pipelining.

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

This 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 $\mathcal{C}$. 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 8.1).

9 Pipelining with Limited Resources

Thus far we have assumed that our 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 6). The change is simple: an operation may not move into a node if the node then violates the resource constraints.

Resource constraints guarantee Property 2 (Lemma 8.3) by imposing a fixed upper bound on the size of program nodes. Thus, the simple rule applies to all loops satisfying Constraint 6.1 without the stronger condition used in Section 8. A proof that Property 1 (Lemma 8.2) holds in the presence of resource constraints may be found in [AN87b].

Figure 9 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 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 loop using Doacross is shown in Figure 10a. An equivalent static SPL schedule is shown in Figure 10b.

We now show how Perfect Pipelining applies to this loop. Figure 11 shows the original loop unwound seven 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 12. 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 12 are equivalent and the start node is equivalent to the first two nodes in the right column. Figure 13 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 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 9: A simple loop L.

(a) Dynamic schedule.

(b) Static SPL program.

Figure 10: Doacross applied to L.

Figure 11: L unwound seven times.

Figure 12: L unwound seven times and compacted.

Figure 13: The same loop after pipelining.

```
move(x , n , m )
if x is an assignment
    then if x ∈ op (s) for all s ∈ succ (n)
        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 14: The Move operator for $\mathcal{D}_{synch}$.

10 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 14). 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 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. The following theorem summarizes the relationship between the three transformations.

Theorem 10.1 For all loops L, $\mathcal{D}_{asynch}L \leq_p \mathcal{D}_{synch}L \leq_p \mathcal{C}u^{\infty}L$.

11 Efficiency

There are loops satisfying Constraint 6.1 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 must 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.

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 other transformations. We believe that Perfect Pipelining will greatly enhance the power of our environment by subsuming the effects of a class of coarse-grain transformations in a uniform, integrated fashion compatible with our fine-grain approach.

13 Acknowledgements

Anne Neirynck and Prakash Panangaden provided a great deal of helpful advice on many aspects of this work. Laurie Hendren, Prakash Panangaden, and Jennifer Widom criticized drafts of this paper and contributed greatly to its final form.

References

[AK84] J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, volume 19, pages 233–246, June 1984.

[AN87a] A. Aiken and A. Nicolau. Loop Quantization: An analysis and algorithm. Technical Report 87-821, Cornell University, March 1987.

[AN87b] A. Aiken and A. Nicolau. Perfect Pipelining: A new loop parallelization technique. Technical Report 87-873, Cornell University, October 1987.

[AN88] A. Aiken and A. Nicolau. A development environment for horizontal microcode. *IEEE Transactions on Software Engineering*, 14(5):584–594, May 1988.

[CCK87] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined architectures. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 295–304, August 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.

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

[Kuc76] D. J. Kuck. Parallel processing of ordinary programs. In *Advances in Computers*, volume 15, 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.