A THEORY OF COMPACTION-BASED PARALLELIZATION

Alexander Aiken

IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, USA

Abstract. We present a simple and uniform transformational system for extracting parallelism from programs. The transformations are studied as a formal system. We define a measure of program improvement, and show that the transformations improve programs with respect to the measure. We show that it is possible to compute limits of infinite sequences of the transformations; this allows our system to implement software pipelining, and leads to a general solution to the problem of software pipelining in the presence of tests.

Using the primitive transformations and the limit-taking transformation, it is possible to express the classical parallelization techniques for vector, multiprocessor, and VLIW machines. Thus, our transformational system can be viewed as a formal foundation for the area of parallelization. We present implementations of two techniques, doacross and a simple form of vectorization.

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 subfield. 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.

We present a system of program transformations for extracting parallelism from programs. This system is intended to be the basis of a practical compiler that generates high-quality code for synchronous machines. This paper, however, focuses on the theoretical issues. At the heart of our approach are four primitive transformations, based on the core transformations of percolation scheduling [29]. The development is geared to studying the properties of the primitive transformations and of perfect pipelining, a transformation that computes limits of infinite sequences of the primitive transformations [5]. Perfect pipelining allows our system to implement software pipelining, a technique for overlapping the iterations of a loop in much the same way that a hardware pipeline overlaps a stream of instructions.
1.1. Loop parallelization

Handling loops well is critical in all compiler optimizations. In compaction-based parallelization, the standard method for extracting parallelism from a loop 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 extract more parallelism, most systems unroll 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. Perfect pipelining overcomes this problem by achieving the effect of unbounded unrolling and compaction of a loop.

We use the program graph shown in Fig. 1 to illustrate perfect pipelining. (The loop control code is simplified 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 statement of an iteration is dependent on the result of the first statement of the previous iteration. Figure 2 shows a schedule after the loop has been unrolled three times and compacted. Statement labels have been substituted for the statements; subscripts indicate the increment to the induction variable. The running time of this loop is $2n$ steps. Figure 3 shows the loop unrolled five times and compacted, in this case

---

1 Two additional memory locations are allocated to each array to handle the extra references generated when $i = n$. Alternatively, the loop bounds can be adjusted and a prologue added as in [9].
the running time is \( \frac{8}{n} \) steps. Note the low parallelism at the beginning and end of the loop body in both of these examples.

Additional unrolling and compaction improves the running time of this loop, but this approach becomes computationally expensive very rapidly. Perfect pipelining derives the program shown in Fig. 4 (in this figure only, the subscripts indicate the absolute value of the array subscripts). Intuitively, perfect pipelining observes that the fourth and fifth nodes of Fig. 3 execute the same statements from different iterations, and that further unrolling and compaction creates more nodes of the same type. We refer to this, informally, as the pattern generated by compaction of the loop.

---

**Fig. 4. Loop after perfect pipelining.**

The loop in Fig. 4 is equivalent to an “infinitely unrolled” and compacted version of the original loop. The transformation achieves continuous (or perfect) pipelining of the loop iterations. The prologue and epilogue to the loop are everything not included in the pattern. For Fortran-like DO loops, it turns out that the perfect pipelining transformation requires little more than rearranging edges between nodes. (Note that the tests in the epilogue are unnecessary and can be deleted; they have been left in to emphasize the relationship between the loops before and after application of perfect pipelining.) The running time of the loop in Fig. 4 is \( n + 3 \) steps.

In this example, the pattern detected by perfect pipelining is very simple because there are no branches (other than exits) in the loop body. Perfect pipelining finds such a pattern on all paths regardless of the flow of control within the loop body.
Perfect pipelining applies in the presence of resource constraints. While it is infeasible to compute optimal loop schedules with resource constraints (even in idealized models [19]), the fact that perfect pipelining achieves continuous overlap of iterations on all paths tends to keep machine utilization high. We prove that the transformation finds a pattern given fixed resources. In Section 6, we provide an example illustrating its performance when the loop has unpredictable flow of control and machine resources are a limiting factor.

Perfect pipelining can be combined with transformations that alter the dependencies of a loop. For example, statement A, in Fig. 2 has subscript \( i + 1 \) instead of \( i \); the indirect reference has been modified to eliminate dependencies caused by incrementing the induction variable. In the final, pipelined loop in Fig. 4, using statement D' compensates for this modification, allowing the loop to exit and “wind-down” without overrunning the bounds of the array. To make our examples realistic, we eliminate dependencies caused by incrementing the induction variable, as in Fig. 2. However, the general use of transformations that change the dependencies of a loop can be treated separately and is beyond the scope of this paper; techniques for including such transformations are discussed in [2, 24].

Finally, perfect pipelining bounds the amount of unrolling required. In previous parallelization systems, particularly trace scheduling [15, 16], there is a trade-off between loop unrolling and exploitable parallelism. As illustrated in Fig. 3, additional unrolling and compaction of a loop body theoretically improves execution time. However, unrolling and compaction increase the size of a program, and machines have storage limitations. In particular, high-performance parallel machines use an instruction cache, and a loop too large for the cache will perform very poorly. Thus, existing systems unroll as much as possible, subject to hardware constraints. In contrast, when a pattern is detected by perfect pipelining, no further unrolling is required. While this pattern may still be too large for an instruction cache, perfect pipelining typically requires much less unrolling and yields very concise schedules. For example, to achieve performance within 10% of the loop in Fig. 4 without perfect pipelining (i.e., by simply compacting an unrolled loop body), the loop in Fig. 1 must be unrolled 30 times. To achieve performance within 1% of the loop in Fig. 4 without perfect pipelining, the loop in Fig. 1 must be unrolled 300 times.

1.2. Overview of the paper

To make our claims precise, we develop a formal account of our transformations. The formalism is powerful enough to capture the intuitive notion of program improvement used informally throughout the literature on parallelization. In Section 2, we define the language to which the transformations apply and provide an operational semantics. In Section 3, a binary relation \( \preceq_p \) is defined on programs using the operational semantics. If \( P \preceq_p P' \) then operations in \( P' \) are executed earlier than corresponding operations in \( P \). In this sense, \( \preceq_p \) measures when one program is “more parallel” than another. We use \( \preceq_p \) to prove that perfect pipelining is better than any finite unrolling with compaction.
Section 4 gives a brief overview of the primitive transformations. Formal definitions and proofs of correctness may be found in [2]. Section 5 defines the perfect pipelining transformation and investigates a number of properties of perfect pipelining. Necessary and sufficient conditions are presented for perfect pipelining to succeed with a given compaction algorithm. An example is developed showing that vectorization of a single statement is an instance of perfect pipelining.

Section 6 develops the simple rule, a simple compaction algorithm that can be used with perfect pipelining. An extended example of loop parallelization with the simple rule is presented; this example illustrates how perfect pipelining applies in the presence of unpredictable flow of control and resource constraints. We use \( \leq_p \) to show that perfect pipelining with the simple rule is at least as general as doacross, a parallelizing transformation for multiprocessors [11]. Other examples of transformations expressible by the primitive transformations and perfect pipelining (such as the wavefront method [28] and trace scheduling [15, 16]) are presented in [2].

2. The model

This section provides the formal foundation for our study of parallelizing transformations. We first informally describe the programming language in which transformations are performed, and then provide a formal definition and an operational semantics. The operational semantics is used to define a relation \( \leq_p \) that measures when one program is "more parallel" than another.

The model of computation and the relation \( \leq_p \) are motivated by synchronous parallel computation and compaction-based parallelization. This model is accurate for synchronous parallel computers, such as pipelined and Very Long Instruction Word machines [17, 18]. In subsequent sections we show that important parallelization techniques for synchronous and asynchronous parallel machines can be expressed using our model and transformational system.

2.1. A simple language

In this section we give an informal description of SPL, a Simple Parallel Language. SPL is an abstraction of the model of computation used to define percolation scheduling [29]. 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; a program is represented by a control flow graph as in Fig. 5a. Each node in the graph contains zero or more primitive statements. These statements 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 statements 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). Statements 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. An example node is given in Fig. 6. The node contains three tests and two assignments. The tests are organized as a tree; assuming that all of the tests evaluate to true, the next node in the execution sequence is E1.

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 [21]. Implementing trees of tests is discussed in [12, 14]. Less general multiway jump mechanisms are used in many horizontal microengines and Multiflow’s TRACE architecture [26].

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 Fig. 5a. Note that this program has only one statement per node; such a program is sequential. Another, more parallel version of the same program is given in Fig. 5b.
2.2. Formal definition of SPL

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 follows the structural style advocated by Plotkin [30].

Figure 7 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 (recall that write conflicts are not permitted). A program is a tuple \( \langle N, n_0, F \rangle \) where:

- \( N \) is a 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 \( \langle B, \text{select}, r, H \rangle \) where:
  - \( B \) is a set of tests,
  - \( \text{select} : B \times \text{Bool} \rightarrow B + H \) is an edge function,
  - \( r \) is the root test or a node name,
  - \( H \) is a set of node names.

As an example, an SPL representation of the node in Fig. 6 is given in Fig. 8. Statement labels are used in the figure to distinguish multiple occurrences of a statement; we assume throughout that all occurrences of a statement are uniquely identifiable.

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 written by statements to model dependency analysis. Dependency analysis

\[
{\{x := 1, y := 1\}, \{a : i < n, b: j - 0, c: j = 0\}, \text{sel}, \{E1, E2, E3, E4\}}
\]

\[
\begin{align*}
\text{Bool} & = \text{tt} + \text{ff} \\
\text{Loc} & = \mathcal{Z} \\
\text{Store} & = \text{Loc} \rightarrow \text{Val} \\
\text{Assign} & = \text{Store} \rightarrow \text{Loc} \times \text{Val} \\
\text{Test} & = \text{Store} \rightarrow \text{Bool}
\end{align*}
\]

\[
\begin{align*}
\text{sel}(a, \text{tt}) & = b & \text{sel}(a, \text{ff}) & = c \\
\text{sel}(b, \text{tt}) & = E1 & \text{sel}(b, \text{ff}) & = \Sigma 2 \\
\text{sel}(c, \text{tt}) & = E3 & \text{sel}(c, \text{ff}) & = E4
\end{align*}
\]
Let $n = (A, C)$, $C = (B, \text{select}, r, H)$, and $A = \{a_i\}$

Constraints on $C$:
- The directed graph induced on $B$ and $H$ by $\text{select}$ is:
  - connected
  - acyclic
  - has a unique root $r$.

Constraint on $A$:
- $\forall s, i, j$ where $i \neq j$, $\text{write}(a_i, s) \cap \text{write}(a_j, s) = \emptyset$

Constraints on $\sim$:
- $\forall n \in N, n_0 \notin \text{succ}(n)$
- $\forall n \in F, \text{succ}(n) = \emptyset$
- $\forall n \in F$ where $n = (A, \langle B, \text{select}, r, H \rangle)$, $A = \{\text{result}\} \land B = \emptyset$

Fig. 9. Program constraints.

determines when two program statements may refer to the same memory location. The analysis is used to determine when it is safe to execute statements in parallel. We define $\text{write}(a, s)$ to be the location written by assignment $a$ in store $s$; $\text{read}(a, s)$ is the set of locations read by assignment (or test) $a$ in store $s$.

In Section 2.1, we discussed well-formedness conditions and semantic constraints not captured by the above description. These requirements are listed in Fig. 9. 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 statement, $\text{result}$, that reads and returns the result of the computation. We assume that $\text{result}$ returns the entire final store, although any set of store locations would do.

2.2.1. Useful functions on program nodes

Figure 10 lists several useful functions on program nodes. We briefly describe each function. $\text{succ}$ returns the immediate successors of a node; when it is convenient

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

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

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

node: Assign + Test $\rightarrow$ Node
$\text{node}(z) = n$ where $z \in \text{op}(n)$

Fig. 10. Functions on program nodes.
we refer to an edge \((m,n)\) instead of writing \(n \in \text{succ}(m)\). \(\text{pred}\) returns the immediate predecessors of a node. The function \(\text{op}\) returns the statements in a node. \(\text{node}(x)\) is the node containing statement \(x\).\(^2\)

### 2.2.2. An operational semantics of SPL

An operational semantics for a language describes the execution of programs written in that language. Figure 11 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: \(\rightarrow\), which defines transitions within a node, and \(\Rightarrow\), which defines transitions between nodes. Each transition between nodes corresponds to one step of a synchronous parallel machine. Rules are read as stating that the assertion below the line holds if the assertions above the line hold.

\[
\begin{align*}
C = \langle B, \text{select}, r, H \rangle, \ t \in B, \ \text{select}(t, t(s)) &= t' \\
\langle C, s, t \rangle \Rightarrow \langle C, s, t' \rangle \\
\end{align*}
\]

\[
\begin{align*}
C = \langle B, \text{select}, r, H \rangle, \ n' \in H \\
\langle C, s, n' \rangle \Rightarrow n'
\end{align*}
\]

\[
\begin{align*}
A = \{ a_i \}, \ a_i(s) = \langle l_i, v_i \rangle, \ s[\ldots, l_i \leftarrow v_i, \ldots] &= s' \\
\langle A, s \rangle \Rightarrow s'
\end{align*}
\]

\[
\begin{align*}
n = \langle A, C \rangle, \ C = \langle B, \text{select}, r, H \rangle, \ n \notin F, \ \langle C, s, r \rangle \Rightarrow n', \ \langle A, s \rangle \Rightarrow s' \\
\langle n, s \rangle \Rightarrow \langle n', s' \rangle
\end{align*}
\]

Fig. 11. The operational semantics of SPL.

The first two rules deal with evaluation of a dag of tests. Consider the first rule. In configuration \(\langle C, s, t \rangle\), if \(t\) is a test, then the next configuration is \(\langle C, s, \text{select}(t, t(t)) \rangle\) if \(t\) evaluates to true in store \(s\); otherwise, the next configuration is \(\langle C, s, \text{select}(t, \text{ff}) \rangle\). The second rule returns the node name that terminates the path selected through the dag of tests—the configuration \(\langle C, s, n' \rangle\) rewrites to \(n'\) if \(n'\) is a node name.

The third rule describes the parallel evaluation of assignments. Each assignment \(a_i\) is evaluated in store \(s\). The resulting pairs \(\langle l_i, v_i \rangle\) represent locations \(l_i\) to be written with new values \(v_i\). (Recall that two assignments cannot write the same location in the same step—thus, the \(l_i\) are distinct.) The configuration \(\langle A, s \rangle\) rewrites to \(s'\), where \(s'\) is \(s\) with the new values written in the appropriate locations.

The first three rules describe the internal (\(\Rightarrow\)) transitions of a node. The final rule describes transitions between nodes. Let \(\langle n, s \rangle\) be a configuration where \(n\) is not a final node. If the evaluation of the dag of tests produces a node \(n'\) and the parallel evaluation of the assignments produces a store \(s'\), then the next configuration is \(\langle n', s' \rangle\).

\(^2\) Again, we assume there is a way of distinguishing between multiple copies of the same statement.
A rewriting sequence is an execution history of the computation of a program. For our purposes, a complete rewriting sequence contains much irrelevant detail; in particular, we are rarely interested in the internal evaluation of a node (the \( \rightarrow \) transitions). The following definition puts a rewriting sequence at the right level of abstraction for viewing execution as transitions from nodes to nodes.

**Definition 2.1 (Execution trace).** The execution trace of program \( P \) in initial store \( s \), written \( T(P,s) \), is the sequence \( \langle n_0, s_0 \rangle \rightarrow \langle n_1, s_1 \rangle \rightarrow \langle n_2, s_2 \rangle \rightarrow \cdots \rightarrow \langle n_k, s_k \rangle \) where \( s_0 = s \), \( n_0 \) is the start node of \( P \), and \( n_k \in F \). Traces are defined only for terminating computations. \(^3\)

Finally, we sometimes need to refer to statements that are actually involved in the internal (\( \rightarrow \)) transitions of a node. To allow us to do this concisely, we define the set of statements executed by a node.

**Definition 2.2.** Let \( \text{exec}(n, s) \) be the set of statements executed by \( n \) in store \( s \). If \( n = \langle A, C \rangle \) where \( C = \langle B, \text{sel}, r, H \rangle \), then

1. if \( x \in A \) then \( x \in \text{exec}(n, s) \),
2. if \( r \in B \) then \( r \in \text{exec}(n, s) \),
3. if \( \langle C, s, r \rangle \rightarrow \langle C, s, t \rangle \) and \( t \in B \) then \( t \in \text{exec}(n, s) \).

### 3. A formalization of parallelism

In this section we develop a formal account of our transformations. This allows us to make precise claims about our transformations and to make comparisons with other transformations. We are interested in transformations that exploit only flow of control and data 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 [8, 23, 33], the wavefront method [28] (or hyperplane method [25]), loop distribution [22], loop interchange [7, 33, 32], trace scheduling [15, 16], and doacross [11].

Our goal is to define a natural relation, independent of any specific transformations, that captures the usual notion of program optimization found in the literature on parallelization. This notion consists of two parts: first, transformations reorder program statements based on dependency information (thus preserving program semantics), and second, a transformed program is better (faster) than the original program. We introduce a preorder on programs, \(" \text{sim} \)" (for similarity), that captures when one program approximates the control and dependency structure of another. We then introduce a relation \( \equiv_p \) that is a restriction of \( \text{sim} \). If \( P \equiv_p P' \), then statements

---

\(^3\) While the results should be extensible to infinite computations, this adds complexity to the formalism and is not important to the issues we address.
in \( P' \) are executed at least as early as corresponding statements in \( P \); in this sense, \( P' \) is more parallel than \( P \).

Informally, a program \( P \) is sim to \( P' \) if \( P' \) executes the same statements as \( P \) in an order compatible with the data and control dependencies present in \( P \). \( P' \) may, however, have additional statements on some paths that do not affect the output of the program. The program in Fig. 5b has more statements on some paths than the program in Fig. 5a, but the two programs are semantically equivalent. A transformation respecting \( \text{sim} \) can only take advantage of control and dependency information computable from a program; other semantic properties of a program are not exploited.

The purpose of \( \text{sim} \) is to establish a sensible correspondence between statements in traces of \( P \) and statements in traces of \( P' \). It is not enough to simply ensure that copies of statements from \( P \) appear in \( P' \), since there is no guarantee that dependencies are preserved.

**Definition 3.1.** 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 \( \langle n_0, s_0 \rangle \mapsto \langle n_i, s_i \rangle \mapsto \langle n_j, s_j \rangle \). Then \( x \prec y \) if \( x \in \text{exec}(n_i), y \in \text{exec}(n_j), \text{write}(x, s_i) \in \text{read}(y, s_j) \), and there is no statement \( z \) in \( n_k \) for \( i < k < j \) such that \( \text{write}(x, s_i) = \text{write}(z, s_k) \).

The relation \( \prec \) models *true* dependencies [23], 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 \( \text{sim} \).

**Definition 3.2 (Similarity).** \( P \) sim \( P' \) if and only if there exists a function \( f \) satisfying:

\[
\forall s \quad x \prec y \text{ in } T(P, s) \Rightarrow f(x) \prec f(y) \text{ in } T(P', s)
\]

\[
\land \quad y' \prec f(x) \text{ in } T(P', s) \Rightarrow f^{-1}(y') \prec x \text{ in } T(P, s)
\]

where \( f \) is one-to-one from statements in \( T(P, s) \) to statements 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 \( P \). To prove that \( \text{sim} \) is a sensible relation, we show that it is sound with respect to the operational semantics.

**Definition 3.3.** The result \( R(P, s) \) of a computation is the value returned by the statement \text{result}; this is the last store of \( T(P, s) \). Two programs \( P \) and \( P' \) are strongly equivalent if \( \forall s \ R(P, s) = s' \iff R(P', s) = s' \).
Two strongly equivalent programs either both compute the same result or both diverge in all possible computations. The relation $\text{sim}$ imposes a much stronger condition: not only must the programs compute the same values, they must compute in "similar" ways. Because we are interested in program transformations that rely on dependency information, our transformations respect $\text{sim}$. We prove that if $P \text{ sim} P'$, then $P$ and $P'$ are strongly equivalent. The following development provides the machinery needed to prove this theorem.

**Definition 3.4.** Consider a trace $T(P, s)$. The trace dependency graph is $D = (V, E)$, where $V$ is the set of all statements in the trace and there is an edge between every pair of dependent statements; i.e., $E = \{(x, y) \mid x < y\}$. Multiple copies of the same statement appearing in the trace are distinct nodes in the trace dependency graph.

The following definition describes the dependency graph of a single statement. $^4$ Lemma 3.6 uses this definition to show that corresponding statements (under $\text{sim}$) have isomorphic dependency graphs. Note that statements with isomorphic dependency graphs compute the same value.

**Definition 3.5.** Given $D = (V, E)$, the trace dependency graph of $T(P, s)$, let $D_x = (V_x, E_x)$, the restriction of $D$ to a statement $x$, be the subgraph induced by all statements on which $x$ directly or indirectly depends. Defined inductively

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

The following lemma shows that any choice of $f$ in Definition 3.2 preserves the dependency structure of $P$ and $P'$.

**Lemma 3.6.** Let $P \text{ sim} P'$. Let $f$ be any mapping demonstrating that $P \text{ sim} P'$. Consider any pair of traces $T(P, s)$ and $T(P', s)$. Let $D_x$ and $D'_{f(x)}$ be the dependency graphs of $x$ in $T(P, s)$ and $f(x)$ in $T(P', s)$ respectively. Then $D_x \cong D'_{f(x)}$.

**Proof.** The proof is by induction on the depth of the graph $D_x$.

**Basis:** The depth of $D_x$ is 1. Then $x$ depends on nothing but the initial store $s$. This implies that $f(x)$ depends on nothing but the initial store, since $y' < f(x)$ in $T(P', s) \Rightarrow f^{-1}(y') < x$ in $T(P, s)$. So $D_x \cong D'_{f(x)}$.

**Step:** Assume that $D_x \cong D'_{f(y)}$ for all statements with dependency graphs of depth $i - 1$ or less. Let $D_x$ have depth $i$. Then $x$ depends on a set of statements $\{y_1, \ldots, y_j\}$ with dependency graphs of depth at most $i - 1$. By the hypothesis, $D_{y_k} \cong D'_{f(y_k)}$. Furthermore, for all $i$, $f(y_j) < f(x)$, and $f(x)$ depends on no other statements because $y' < f(x)$ in $T(P', s) \Rightarrow f^{-1}(y') < x$ in $T(P, s)$. So $D_x \cong D'_{f(x)}$. $\square$

$^4$ This is related to the notion of a slice in static analysis of programs [31].
Theorem 3.7 (Soundness of $\text{sim}$). If $P \text{ sim } P'$, then $P$ is strongly equivalent to $P'$.

Proof. Let $f$ be the dependency-preserving mapping used to show $P \text{ sim } P'$. Consider any pair of traces $T(P, s)$ and $T(P', s)$. The statement $\text{result}$ is distinguished and occurs only once in each trace; therefore, $f$ maps $\text{result}$ in $T(P, s)$ to $\text{result}$ in $T(P', s)$. By Lemma 3.6, $D_{\text{result}} = D'_{\text{result}}$. The statement $\text{result}$ returns the final store, so the final store of the two traces is the same. Because the choice of traces was arbitrary, we conclude that the two programs always have the same final store. Therefore, $\forall s \; R(P, s) = s' \Rightarrow R(P', s) = s'$. □

We now introduce the relation $\leq_p$. If $P \leq_p P'$, then all statements in $P'$ are executed at least as early in the execution trace as corresponding statements in $P$.

Definition 3.8. Let $x \in \text{exec}(n_x)$ in $T(P, s)$. The position of $x$, written $\text{pos}(x)$, is $i$.

$$P \leq_p P' \Rightarrow \exists f \; s.t. \; P \text{ sim } P' \land \forall s \; \text{pos}(x) = i \text{ in } T(P, s) \Rightarrow \text{pos}(f(x)) = j \text{ in } T(P', s) \text{ where } i \geq j.$$ 

The purpose of $\leq_p$ is to measure improvement: if $P \leq_p P'$ then $P'$ should be a better program than $P$. Ideally, $\leq_p$ would be a partial order. A partial order is reflexive ($P \leq_p P$), transitive ($P \leq_p P' \land P' \leq_p P'' \Rightarrow P \leq_p P''$), and antisymmetric ($P \leq_p P' \land P' \leq_p P \Rightarrow P = P'$). It is a straightforward to verify that $\leq_p$ is reflexive and transitive. It is not, however, antisymmetric. A counter example is given in Fig. 12. In this example, both programs have the same traces, so $P \leq_p P'$ and $P' \leq_p P$. However, the two programs are different.

The problem arises because $\leq_p$ is based on behavioral or runtime properties—how the program executes—rather than the syntactic representations of programs. In what sense, then, does $\leq_p$ measure parallelism if two distinct programs can each be better than the other? The following theorem shows that if $P \leq_p P'$ and $P' \leq_p P$, then $P$ and $P'$ are indistinguishable in the operational semantics. In every (finite) computation, at each time step, $P$ and $P'$ execute the same statements—thus neither is “more parallel” than the other.

Fig. 12. Programs where $P \leq_p P'$ and $P' \leq_p P$. 
Theorem 3.9. If \( P \preceq_p P' \) and \( P' \preceq_p P \), then \( P \) and \( P' \) are behaviorally equivalent; that is,

\[
\forall T(P, s), T(P', s) \quad \text{exec}(n_i, s_i) = \text{exec}(n'_i, s'_i)
\]

where \( \langle n_i, s_i \rangle \) is the \( i \)-th step of \( T(P, s) \) and \( \langle n'_i, s'_i \rangle \) is the \( i \)-th step of \( T(P', s) \).

Proof. Assume \( P \preceq_p P' \land P' \preceq_p P \). Let \( T(P, s) = \langle n_0, s_0 \rangle \rightarrow \langle n_1, s_1 \rangle \rightarrow \cdots \) and let \( T(P', s) = \langle n'_0, s'_0 \rangle \rightarrow \langle n'_1, s'_1 \rangle \rightarrow \cdots \). Let \( f \) and \( g \) be functions used to show \( P \preceq_p P' \) and \( P' \preceq_p P \), respectively. The proof is by induction on the length of \( T(P, s) \).

**Basis:** Consider \( n_0 \) and \( n'_0 \). Let \( x \) be a statement in \( \text{exec}(n_0) \). From the definition of \( \preceq_p \), \( \text{pos}(f(x)) \leq 0 \), so \( \text{pos}(f(x)) = 0 \). A symmetric argument shows that statements in \( \text{exec}(n'_0) \) must be mapped to statements in \( \text{exec}(n_0) \) under \( g \). Because there is a one-to-one mapping from statements in \( n_0 \) to statements in \( n'_0 \) and vice versa (and therefore an onto mapping), we conclude that \( \text{exec}(n_0) = \text{exec}(n'_0) \). This implies that \( s_1 = s'_1 \).

**Step:** Assume, for all \( j \leq i \), that \( s_j = s'_j \). Assume further that \( f \) is one-to-one and onto from statements in the first \( i-1 \) steps of \( T(P, s) \) to statements in the first \( i-1 \) steps of \( T(P', s) \), and that \( g \) is one-to-one and onto from statements in the first \( i-1 \) steps of \( T(P', s) \) to statements in the first \( i-1 \) steps of \( T(P, s) \).

Consider the \( i \)th step of \( T(P, s) \). Let \( x \in \text{exec}(n_i) \). By definition of \( \preceq_p \), \( \text{pos}(x) = \text{pos}(f(x)) \). But, by the hypothesis, \( f \) is one-to-one and onto in the first \( i-1 \) steps of \( T(P, s) \) and \( T(P', s) \). Therefore, \( \text{pos}(x) = \text{pos}(f(x)) \), which shows that \( f(x) \in \text{exec}(n'_i) \). A symmetric argument shows that statements in \( \text{exec}(n'_i) \) must be mapped to statements in \( \text{exec}(n_i) \) under \( g \). As in the basis step, this implies that \( \text{exec}(n_i) = \text{exec}(n'_i) \) and that \( s_{i+1} = s'_{i+1} \).

Let \( =_p \) be the equivalence relation induced by \( \preceq_p \) (i.e., \( P =_p P' \iff P \preceq_p P' \land P' \preceq_p P \)). Theorem 3.9 justifies \( \preceq_p \) as a measure of parallelism by showing that programs indistinguishable in \( \preceq_p \) are indistinguishable in their runtime parallelism. In addition, if \( P \preceq_p P' \) and \( P \neq_p P' \), then \( P' \) is strictly better than \( P \) in the sense that some statement is executed earlier in some trace of \( P' \) than in the corresponding trace of \( P \). We write \( P <_p P' \) if \( P \preceq_p P' \) and \( P \neq_p P' \). Note that \( P <_p P' \) does not

---

![Diagram](image)

**Fig. 13.** \( P \preceq_p P' \) but the execution times are the same. (a) A program \( P \); (b) a program \( P' \).

**Fig. 14.** The delete transformation.
Theory of compaction-based parallelization

imply $P'$ actually has a shorter running time than $P$; it simply shows that a statement is executed earlier. For example, program $P'$ in Fig. 13a executes no faster than program $P$ in Fig. 13b, but $P \preceq P'$. The relation $\preceq$ is motivated by the compaction paradigm—transformations move statements earlier in the schedule, although that alone does not guarantee an improvement in running time. It is certainly the case, however, that scheduling statements earlier cannot lengthen execution time.

**Lemma 3.10.** Let $|T(P, s)|$ be the number of steps in $T(P, s)$. If $P \preceq P'$, then $|T(P, s)| \geq |T(P', s)|$.

**Proof.** Immediate from the definition of $\preceq$ (Definition 3.8).

3.1. Other relations

The relation $\preceq$ is not the only possible measure for comparing parallelism in programs. An alternative is to use the ordering induced by some set of program transformations—that is, $P \preceq P'$ if $P$ can be transformed into $P'$. It is fairly easy to obtain a partial order in this way. In fact, the set of primitive transformations we use induces a partial order on programs. We rejected this approach because nothing can be said about transformations not directly expressible with the chosen primitives. In Section 5, we introduce transformations that cannot be implemented using our primitives; $\preceq$ is crucial to proving properties of these transformations. With $\preceq$, there is an independent formalism in which transformations can be compared.

4. The core transformations

The core transformations are the building blocks of our transformational system. 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. The transformations presented here are based upon the core transformations of percolation scheduling [29]. An informal description of each transformation is presented; formal definitions and proofs of correctness are in [2]. Discussion of the efficient implementation of the core transformations is in [3, 2].

4.1. An overview of the transformations

There are four primitive transformations: delete, move-op, move-test, and unroll. Code motions are accomplished with move-op (for moving assignments) and move-test (for moving tests). Delete improves a program's running time by removing empty nodes. Unroll (loop unrolling) aids parallelization by increasing the number of statements available to other transformations.

An illustration is given for each transformation. Only the relevant portion of the program graph is shown; incoming edges are denoted by $I_j$ and exiting edges by $E_j$. 
4.1.1. Delete

The delete transformation removes a node from the program graph if it is empty (contains no statements) or unreachable. A node may become empty or unreachable as a result of other transformations. In Fig. 14, assume $n$ is an empty node. When $n$ is removed from the program graph, $n$’s predecessors are changed to point to $n$’s successor. Note that an empty node has exactly one successor.

4.1.2. Move-op

The move-op 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 statements of $m$, and $x$ does not write any storage location live at $m$. (A storage location is live at $m$ if it could possibly be read in some continuation of a computation from $m$ [1].)

Care must be taken not to affect the computation of paths passing through an $n_j$ but not through $m$. To ensure this, the original node $n_j$ is preserved on the other paths. An illustration of move-op applied to a single assignment $x$ is given in Fig. 15; an illustration of move-op applied to several assignments is given in Fig. 16.

While move-op is defined to move any subset of copies of an assignment that can legally move from a set of nodes to a common predecessor, we assume that move-op always moves all copies. Move-op can also be defined to allow distinct

![Fig. 15. The move-op transformation applied to one assignment.](image1)

![Fig. 16. The move-op transformation applied to several assignments.](image2)

\(^5\) This transformation combines the primitives move-op and unify of [29].
statements that are mutually dependent to move into a node $m$ [29], for simplicity, we do not consider this possibility.

4.1.3. Move-test

The move-test transformation moves a single copy of a set of identical tests $\{x_j\}$ from a node $n$ to a node $m$ through an edge $(m, n)$ provided that no dependency exists between $x$ and the statements of $m$. Paths passing through $n$ but not through $m$ must not be affected; as for move-op, $n$ is preserved on the other paths. Because we allow an arbitrary rooted dag of tests in a node and the tests being moved may come from arbitrary points 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 the $x_j$. In $n_t$, the tests on $x_j$'s true branch are inserted in $x_j$'s place; in $n_f$, the tests on $x_j$'s false branch are inserted in $x_j$'s place. Figure 17 illustrates the transformation applied to a single test. In the figure, $a$ 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. While move-test is defined to move any subset of copies of a test from a node $n$ to a predecessor $m$, we assume that move-test always moves all copies.

![Fig. 17. The move-test transformation.](image)

4.1.4. Unrolling

Loop unrolling (or unwinding) is a standard nonlocal transformation [1]. 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 statements (the unrolled loop body) for scheduling. The statements in the unrolled loop body come from previously separate iterations and are thus freer of the order imposed by the original loop. An illustration of a loop unrolled once is given in Fig. 18. We use the shorthand $u^iL$ for loop $L$ unrolled $i$ times, where $u^1L = L$. The unrolled iterations of $u^iL$ are $L_0, \ldots, L_{i-1}$. 
4.2. Correctness

A transformation \( \mathcal{T} \) is correct if for any program \( P \), \( \mathcal{T}(P) \) is strongly equivalent to \( P \). The following theorem justifies the core transformations by showing that they are correct and that some improvement results from their application.

**Theorem 4.1.** Let \( \mathcal{T} \) be delete, move-op, or move-test. Then for all \( P, 3 \leq_p \mathcal{T}(P) \). Let \( \mathcal{T} \) be unrolling. Then for all \( P, P \equiv_p \mathcal{T}(P) \).

**Proof.** The transformations preserve dependencies and do not remove a statement from any path on which it occurs—thus \( P \text{ sim } \mathcal{T}(P) \). For each of delete, move-op, or move-test, if it succeeds, at least one statement appears earlier on at least one path, so \( P \leq_p \mathcal{T}(P) \). For unrolling, for any initial state \( s \), \( T(P, s) = T(\mathcal{T}(P), s) \) or both traces are undefined, so \( P \equiv_p \mathcal{T}(P) \). \( \square \)

5. Perfect pipelining

In this section, we develop the perfect pipelining algorithm. Perfect pipelining resides outside of the basic transformational system discussed in Section 4; no sequence of core transformations can achieve the loop in Fig. 4 from the loop in Fig. 1. We use \( \leq_p \) to establish the relationship between the core transformations and perfect pipelining.

5.1. Formalization

We make precise the informal notion that a loop transformed by perfect pipelining is an infinitely unrolled and compacted version of the original loop. We assume that a program is a simple (innermost) loop. (Techniques for handling nested loops are presented in [2, 4].) Let \( \mathcal{C}u' L \) stand for a compaction algorithm \( \mathcal{C} \) applied to
We give a method, for the class of compaction algorithms discussed below, to compute a program $Cu^xL$ satisfying

$$\forall i \quad Cu^iL \leq_p Cu^xL.$$  

5.2. Conditions for perfect pipelining

It is intuitively obvious that the loop in Fig. 4 is the “right” pipelined version of the loop in Fig. 3. The loop in Fig. 3 contains the first instance of a pattern which is repeated with additional unrolling and compaction; the loop in Fig. 4 is a closed form of this pattern. However, it is not obvious how a compiler can safely detect this pattern by examining the finite number of loop iterations exposed by unrolling. Consider the following analogy with numbers:

Answer the following question: I have a number in mind; the first five digits of its decimal expansion are 0.99999. What is the next digit? This question cannot be answered since there are many numbers that begin 0.99999. Without additional information, no repeating behavior can be inferred.

The “additional information” is provided by restricting the compaction algorithms $C$ that can be used with perfect pipelining. There are two basic requirements that any compaction algorithm used with perfect pipelining must satisfy. First, $Cu^{i+1}L$ must be a better approximation to the final (pipelined) loop than $Cu^iL$; i.e., the compaction algorithm must be monotonic. Second, the compaction algorithm must exhibit some repeating behavior.

**Definition 5.1 (Well-behaved).** Let $C$ be a deterministic compaction algorithm built from move-op, move-test, and delete. Consider the sequence $Cu^1L, Cu^2L, Cu^3L, \ldots$. If $\forall i \quad Cu^iL \leq_p Cu^{i+1}L$, then $C$ is well-behaved on $L$ and the $Cu^iL$ form a chain. (A chain is a set of programs totally ordered by $\leq_p$.)

This is the monotonicity condition—the compaction algorithm should perform better with larger unrollings of the loop. For the moment, we require that a compaction algorithm be well-behaved. This requirement is actually stronger than necessary; we explain why and develop a better monotonicity condition in Section 5.6.

To formalize the second requirement, we define equivalence of program nodes. Nodes $n$ and $n'$ are equivalent if, in a large enough unrolled and compacted program, starting computation at $n$ in a store $s$ is indistinguishable from starting computation at $n'$ in store $s$. A compaction algorithm $C$ can be pipelined if $C$ generates finitely many distinct classes of equivalent nodes. The definition of equivalent nodes requires some notation; to discuss the “same” node in $u^iL$ and $u^{i+j}L$.

**Definition 5.2.** A node $n$ in $L$ and a node $n'$ in $L'$ are equal, written “$n$ in $L = n'$ in $L'$”, if all acyclic paths from the start node to $n$ and $n'$ are identical in both programs.
Definition 5.3 (Equivalent nodes). Two program nodes \( n \) and \( n' \) in \('Cu^i L\) are equivalent if, for any integer \( k \), there is an unrolling \( j \) such that in \('Cu^{i+j} L\):
\[
\langle n_1, s \rangle \rightarrow \langle n_2, s_2 \rangle \rightarrow \cdots \rightarrow \langle n_k, s_k \rangle = \langle n'_1, s \rangle \rightarrow \langle n'_2, s_2 \rangle \rightarrow \cdots \rightarrow \langle n'_k, s_k \rangle
\]
where \( n \) in \('Cu^i L = n_1 \) in \('Cu^{i+j} L \) \& \( n' \) in \('Cu^i L = n'_1 \) in \('Cu^{i+j} L \).

For example, in Fig. 3, nodes four and five are equivalent with \( j = k + 5 \). Unless otherwise stated, we assume that compaction algorithms are well-behaved and generate only finitely many classes of equivalent nodes for a given loop.

5.3. The algorithm

If nodes \( n \) and \( n' \) can be identified as equivalent, then perfect pipelining may delete \( n' \) and add edges from \( \text{pred}(n') \) to \( n \). For example, in Fig. 3 the third and fourth nodes are equivalent: for a sufficiently large unrolling with compaction, computations beginning at the two nodes appear identical.

Lemma 5.4 below establishes an important property of well-behaved compaction algorithms when machine resources are limited. While resource limits can be arbitrarily complex, we simply impose a limit \( c \) on the number of statements that a node can contain. We assume that the core transformations are modified to move statements only if the resulting program graph does not violate the resource constraints.

Lemma 5.4. Let \( C \) be well-behaved on \( L \). For every depth \( k \) there is an unrolling \( i \) such that node \( n \) at depth \( k \) is unchanged for any unrolling greater than \( i \). That is,
\[
\exists i \text{ s.t. } \forall j > i \quad n \text{ in } 'Cu^i L = n \text{ in } 'Cu^j L.
\]

Proof. Because \( C \) is well-behaved, a statement \( x \) in \( \text{op}(n) \) in \('Cu^i L \) can only be scheduled in \( n \) or a predecessor of \( n \) in \('Cu^j L \). The lemma follows immediately from the fact that resources are bounded. \( \Box \)

The following theorem establishes a key fact about the compaction algorithms under consideration (those that are well-behaved and generate only finitely many distinct classes of equivalent nodes).

Theorem 5.5 (Convergence). For a sufficiently large unrolling \( i \), on every path in \('Cu^i L \) there are two equivalent nodes \( n \) and \( n' \) where \( n \) and \( n' \) do not change with greater unrolling and compaction.

Proof. Because resources are bounded, all paths must grow longer with larger unrollings. Therefore, because there are only finitely many classes of equivalent nodes, for a large enough unrolling every path has at least two equivalent nodes. The second clause of the theorem is guaranteed by Lemma 5.4. \( \Box \)
let \( i \) be the number in Theorem 5.5
\[ P := C^iL; \]
for each path \( p \) through \( P \) do
begin
- let \( n \) and \( n' \) be equivalent nodes on \( p \);
- Replace edges \((m, n')\) by \((m, n)\);
- Delete any unreachable nodes.
end;

Fig. 19. Perfect pipelining.

The perfect pipelining transformation is given in Fig. 19. 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 clarity, we have presented the simplest possible form of Theorem 5.5 and the perfect pipelining algorithm. A stronger statement actually holds: it is not necessary that the equivalent nodes be on the same path—it is sufficient to find equivalent nodes anywhere in the loop body. Exploiting this property speeds convergence of the algorithm in practice.

Refer again to the loop in Fig. 1. In Fig. 3, the loop has been unrolled five times and compacted. (The compaction algorithm used in this example is presented in Section 6.2.) The fourth and fifth nodes are equivalent. The transformation deletes the fifth node and adds an edge from the fourth node to itself with an induction variable increment of one. The result of applying perfect pipelining to this loop (combined with minor adjustments to account for altering dependencies of the original loop; see Section 1.1) is shown in Fig. 4.

5.4. More on convergence

By Theorem 5.5, a sufficient condition for convergence of the perfect pipelining algorithm is that the compaction algorithm generates finitely many classes of equivalent nodes. This is also a necessary condition: if a compaction algorithm \( C \) generates infinitely many classes of equivalent nodes, then it cannot be that there are two equivalent nodes on every path in any \( C^iL \). In other words, if the compaction algorithm generates infinitely many classes of equivalent nodes, then there is no finite representation of \( C^\infty L \). Thus, a well-behaved compaction strategy converges if and only if it generates finitely many classes of equivalent nodes.

Checking that a compaction algorithm generates only finitely many classes of equivalent nodes can be difficult. In this section, we study conditions that are easy to verify and that provide some insight into the important properties of those compaction algorithms that can be pipelined.

A necessary condition for convergence is a bounded compaction algorithm. Intuitively, when a loop is compacted with a bounded compaction algorithm, every statement in the compacted loop is executed at the same rate regardless of the unrolling chosen. In the following definition, let the distance between two nodes \( n \) and \( n' \) be the length of the longest path from \( n \) to \( n' \) that does not include a backedge. If there is no such path, the distance is undefined.
Definition 5.6 (Boundedness). Let $x_i$ denote the statement $x$ in unrolled iteration $L_i$. A compaction algorithm is bounded for loop $L$ if for all statements $x$ and $y$ and integers $j$, there is a constant $k$ such that for all $i$ and $h$ the distance between $node(x_i)$ and $node(y_{i+j})$ in $\mathcal{C}u^hL$ is at most $k$ (or undefined).

Consider the loop in Fig. 20. The same loop unrolled five times and compacted maximally—everything moved as far as possible using the core transformations—is shown in Fig. 21. We assume that up to six statements can be executed in parallel. Maximal compaction is not a bounded compaction algorithm for this loop; note that the first statement vectorizes while the second is involved in a recurrence. If resource constraints allow more than two statements to be executed in parallel, then the distance between statements $A_i$ and $B_i$ grows arbitrarily large as the loop is unrolled and compacted. In terms of Definition 5.3, maximal compaction generates an unbounded number of classes of equivalent nodes. In this example, for any unrolling with maximal compaction, every node is equivalent only to itself.

A bounded compaction algorithm is a necessary—but not sufficient—condition for convergence of perfect pipelining. The compaction algorithm in Fig. 22 is bounded, but it also generates an unbounded number of classes of equivalent nodes. This compaction algorithm uses global information about the loop (the number of the iteration being compacted) to prevent any pattern from emerging. Thus, a compaction algorithm that converges should use only local information. For example, the compacted form of an iteration should depend only on a finite number of adjacent iterations. A local compaction algorithm cannot use global analysis of the loop to decide where to move a given statement. The ideas of locality and boundedness in compaction algorithms are formalized by the following theorem.

**Fig. 20.** Another sample loop.

for $i := 1$ to $n$ do
begin
    B: $B[i] := B[i - 1]$;
end;

**Fig. 21.** The loop unrolled and compacted with maximal compaction.

**Fig. 22.** An irregular compaction algorithm.
Definition 5.7. A finite automaton \( M \) is a machine with a one-way read-only input tape, a one-way write-only output tape, and a finite internal state. At each step, \( M \) may read one symbol from the input tape, write one symbol on the output tape, and make an internal state transition.\(^6\)

Theorem 5.9. Perfect pipelining converges with compaction algorithm \( \mathcal{C} \) if, for a given loop \( L \), \( \mathcal{C} \) is well-behaved and \( \mathcal{C} \) acts as a finite automaton.

Proof. Because \( \mathcal{C} \) has a finite control, it has at most a finite number of distinct states. A classical result of automata theory shows that a finite-state machine must loop (repeat states) on sufficiently large inputs \([20]\). Thus, a compaction algorithm with a finite state must produce equivalent nodes on all paths for a sufficiently large unrolling. \( \square \)

Locality limits the size of the state. In Fig. 22, the compaction algorithm computes \( \pi \) to arbitrary precision, so it is not a finite-state compaction algorithm. An unbounded compaction algorithm also has unbounded state: no finite-state machine can generate more than a finite number of distinct classes of equivalent nodes. There are, of course, compaction algorithms with unbounded state which converge. Thus, Theorem 5.8 provides a sufficient, but not necessary, condition for convergence.

5.5. Correctness of perfect pipelining

We prove that perfect pipelining is correct and that it is equivalent to infinite unrolling and compaction using a compaction algorithm \( \mathcal{C} \). The following technical lemmas simplify the proof. Recall that \( T(P, s) \) is the execution history of program \( P \) in initial store \( s \) (Definition 2.1).

Lemma 5.9. Let \( n \) and \( n' \) be equivalent nodes in \( \mathcal{C} u^1 L \). Let \( \langle n, s \rangle \rightarrow \langle m, s' \rangle \) and \( \langle n', s \rangle \rightarrow \langle m', s' \rangle \), where for all \( j \), there exist \( m_o \) and \( m'_o \) such that

\[
\begin{align*}
\text{m in } \mathcal{C} u^j L &= m_o \text{ in } \mathcal{C} u^{i+j} L, \\
\text{m' in } \mathcal{C} u^j L &= m'_o \text{ in } \mathcal{C} u^{i+j} L.
\end{align*}
\]

Then \( m \) and \( m' \) are equivalent.

Proof. Immediate by definition of equivalence (Definition 5.3). \( \square \)

Lemma 5.10. Let \( \mathcal{C} u^\infty L \) be the result of perfect pipelining. For any integer \( k \), there is an unrolling \( i \) such that \( T(\mathcal{C} u^\infty L, s) \) is identical to \( T(\mathcal{C} u^i L, s) \) for the first \( k \) steps.

Proof. Choose an \( i \) such that every node on a path of length \( k \) or less from the start node in \( \mathcal{C} u^i L \) is unchanged with larger unrollings and compaction. (The

\( ^6 \) This is actually a special variety of finite automaton, the Mealy machine \([20]\).
existence of \( i \) is guaranteed by Lemma 5.4.) We proceed by induction on the number of times \( T(\mathcal{C}u^iL, s) \) crosses a backedge introduced by the perfect pipelining algorithm.

**Basis:** There are two cases: \( T(\mathcal{C}u^\infty L, s) \) crosses no edges introduced by the algorithm for \( k \) steps, and only the \( k \)th step of \( T(\mathcal{C}u^\infty L, s) \) crosses an edge introduced by the algorithm. In case one, the traces are identical for \( k \) steps by Lemma 5.4 and the choice of unrolling in the algorithm. In case two, assume that \( \langle n_0, s \rangle \Rightarrow \langle n_{k-1}, s_{k-1} \rangle \) in both programs. Let \( \langle n_{k-1}, s_{k-1} \rangle \Rightarrow \langle n_k, s_k \rangle \) in \( \mathcal{C}u^iL \) and let \( \langle n_{k-1}, s_{k-1} \rangle \Rightarrow \langle n, s_k \rangle \) in \( \mathcal{C}u^\infty L \), where \( \langle n_{k-1}, n \rangle \) is an edge introduced by the algorithm. Let \( h \) be the unrolling used in the perfect pipelining algorithm. The algorithm replaced edge \( \langle n_{k-1}, n' \rangle \) with \( \langle n_{k-1}, n \rangle \), where \( n' \) and \( n \) are equivalent nodes in \( \mathcal{C}u^hL \). By Lemma 5.4, \( n \) in \( \mathcal{C}u^hL = n \) in \( \mathcal{C}u^iL \). But then there is an \( m \) in \( \mathcal{C}u^iL \) such that \( m \) is equivalent to \( n_{k-1} \) and \( n \) in \( \mathcal{C}u^\infty L = m \) in \( \mathcal{C}u^\infty L \). So \( n = n_{k-1} \).

**Step:** Assume that \( \langle n_0, s \rangle \Rightarrow \langle n_{k-1}, s_{k-1} \rangle \) in both programs, and let \( \langle n_{k-1}, s_{k-1} \rangle \Rightarrow \langle n, s_k \rangle \) in \( \mathcal{C}u^\infty L \). Furthermore, assume that there is a node \( m \) such that \( m \) in \( \mathcal{C}u^iL = n_{k-1} \) in \( \mathcal{C}u^\infty L \) and \( m \) is equivalent to \( n_{k-1} \). The rest of the proof is similar to the base case using Lemma 5.9. \( \square \)

**Theorem 5.11.** For all \( i \) and \( L \) for which \( \mathcal{C} \) is well-behaved and generates only finitely many classes of equivalent nodes, \( \mathcal{C}u^iL \leq_p \mathcal{C}u^\infty L \).

**Proof.** Let \( k \) be the length of \( T(\mathcal{C}u^iL, s) \). By Lemma 5.10, there is a program \( \mathcal{C}u^iL \), where \( j \geq \max(i, k) \), such that \( T(\mathcal{C}u^iL, s) = T(\mathcal{C}u^\infty L, s) \). Because \( \mathcal{C} \) is well-behaved, \( \mathcal{C}u^iL \leq_p \mathcal{C}u^iL \). We conclude that \( \mathcal{C}u^iL \leq_p \mathcal{C}u^\infty L \). \( \square \)

Theorem 5.11 shows that perfect pipelining is as good as full unrolling and compaction on all paths. The transformation computes a closed form of the pattern generated by repeated unrolling and compaction using \( \mathcal{C} \).

**5.6. More on monotonicity**

As mentioned in Section 5.2, the original definition of a well-behaved compaction algorithm is too strong. While the results of the previous sections are true for any well-behaved compaction algorithm, there are interesting compaction algorithms that converge but are not well-behaved. Consider Fig. 23. We assume that at most two statements can be executed in parallel due to resource constraints. We use maximal compaction (\( \mathbb{S} \)) as the compaction algorithm for this example. Because the loop body has only a single statement and no dependencies, maximal compaction is a bounded and local compaction algorithm for this loop. Figure 23b shows \( \mathcal{G}u^2L \), \( \mathcal{G}u^3L \), and \( \mathcal{G}u^4L \). Note that \( \mathcal{G}u^2L \neq_p \mathcal{G}u^3L \) and \( \mathcal{G}u^2L \equiv_p \mathcal{G}u^4L \equiv_p \mathcal{G}u^\infty L \). Perfect pipelining converges with \( \mathcal{G}u^4L \) because the two nodes are equivalent.

When resources are a limiting factor, a loop unrolled and compacted \( i \) times may be faster than a loop unrolled and compacted \( i + j \) times because the smaller loop fits the resources better. The purpose of this section is to extend our theory to cover
all interesting compaction algorithms not satisfying the constraints of previous sections. To do this, we weaken the definition of a well-behaved compaction algorithm.

**Definition 5.12.** $C$ is well-behaved if $\forall i, k \ C^i u^L \leq_p C^{2k} u^L$.

Definition 5.12 only requires that $C^{2k} u^L$ be at least as good as duplicating the loop body of $C^k u^L$, so this is quite a weak constraint. The definition of equivalent nodes must be changed to reflect the change in the definition of well-behaved.

**Definition 5.13.** Two program nodes $n$ and $n'$ in $C^i u^L$ are equivalent if, for any integer $k$, there is an integer $j = 2^c$ such that in $C^j u^L$,

$$\langle n_1, s_1 \rangle \rightarrow \langle n_2, s_2 \rangle \rightarrow \cdots \rightarrow \langle n_k, s_k \rangle = \langle n'_1, s_1 \rangle \rightarrow \langle n'_2, s_2 \rangle \rightarrow \cdots \rightarrow \langle n'_k, s_k \rangle$$

where $n$ in $C^i u^L = n_i$ in $C^j u^L$ and $n'$ in $C^i u^L = n'_i$ in $C^j u^L$.

All of the results of the previous sections hold using Definitions 5.12 and 5.13; for any chain $C^0 u^L, C^1 u^L, \ldots$ where $C^i u^L \leq_p C^{i+1} u^L$, perfect pipelining computes a loop $C^x u^L$ such that $\forall j \ C^j u^L \leq_p C^x u^L$. However, because the definition of well-behaved is so weak, the result of perfect pipelining using different chains may be different; $C^x u^L$ may not be unique. Let the canonical chain be $C^0 u^L, C^1 u^L, C^2 u^L, C^4 u^L, C^8 u^L, \ldots$. We define $C^x u^L$ to be the result of perfect pipelining using the canonical chain.

Using this extended definition of well-behaved, our system can express a limited form of vectorization. Consider again Fig. 23. The original loop in this example consists of a single statement with no data dependencies. The loop $G^{x} u^L$ executes $k$ statements at a time, where $k$ is determined by the resource constraints of the machine. This is also what vectorization achieves on this example. Thus, vectorization of a single statement can be viewed as the limit of an infinite sequence of core transformations.
6. Perfect pipelining: a case study

In this section, we develop one compaction algorithm for use with perfect pipelining, the simple rule. We use this algorithm to illustrate how perfect pipelining applies to loops with arbitrary flow of control. We compare the effect of perfect pipelining using the simple rule with a standard multiprocessor parallelization technique, doacross [10]. Other compaction strategies for use with perfect pipelining may be found in [2].

6.1. The dependencies

A loop-carried dependency [6] is a dependency between separate iterations of a loop. In this context we are referring to the approximate dependencies a compiler computes using conservative analysis, rather than the precise trace dependency graphs used to define $\leq_p$ (Definition 3.1). We consider single (nonnested) loops satisfying the following constraint for any unrolling.

Constraint 6.1. Assume there is a dependency between statement $x$ and $y$ in $L$. If the dependency is loop-carried, then in $u^jL$ there is a dependency between statement $x$ of $L_j$ and $y$ of $L_{j+1}$ for all $j$. If the dependency is loop-independent, then in $u^jL$ there is a dependency between statement $x$ of $L_j$ and $y$ of $L_j$ for all $j$.

The constraint is quite restrictive; even so, most loops arising in practice can be automatically rewritten to satisfy it [27].

6.2. The simple rule

To simplify the algorithms, the primitives move-op and move-test are combined into one transformation move (Fig. 24). The simple rule moves an iteration $L_j$ as far “up” in the program graph on as many paths as possible. In the simple rule, statements in the iteration remain in adjacent nodes and the iteration keeps its “shape”—statements appear in the order of the original loop body.

The basic step of the simple rule is to move each statement in one copy of an iteration up one node in the program graph. An algorithm that accomplishes this is given in Fig. 25. We assume that statements are identified with their unrolled iteration $L_j$. A fail command causes the entire recursive computation to terminate and restores the original program graph.

```
procedure move($z, n, m$)
  if $z$ is an assignment
    then $P :=$ move-op($P, z, move~set, m$)
      where move-set contains every member of $\text{succ}(m)$ with
      a copy of $z$ that can move
    else $P :=$ move-test($P, move~set, n, m$)
      where move-set contains all copies of $z$ in node($x$);

Fig. 24. The move transformation.
```
procedure move-iteration(x, n, m)
begin
  if x ∈ op(n)
  then
    begin
      move(x, n, m);
      if move failed then fail
    end;
  (* next-op-in-il(x, n, p) is next statement in the iteration
    after x on edge (n, p). *)
  for each (p, y) s.t. p ∈ succ(n) ∧ next-op-in-il(x, n, p) = y do
    move-iteration(y, p, n);
  Delete all empty nodes.
end;

Fig. 25. Moving an iteration.

(* Let P = u L *)
for each iteration L₀, ..., Lᵢ do
begin
  X := {x} where x is the first statement in Lᵢ;
  repeat
    (* we assume that X always contains all copies of statement x *)
    Select z ∈ X such that z's iteration can move and
    the depth of m in the graph is maximized;
    move-iteration(z, node(z), m)
  until no iteration can move;
  Delete all empty nodes.
end;

Fig. 26. The simple rule.

The simple rule is given in Fig. 26. As many statements are combined as possible, thus minimizing code explosion. As iterations move through the program graph, copies of statements—forming distinct copies of the iteration—are generated where paths split. The top-level algorithm refers to the first statements in each copy of the iteration; the other statements are handled by move-iteration.

In the remainder of this paper, we use ∈ for the simple rule. An important property of ∈ is that it is maximal—for any ∈' using move-iteration and for all programs P and unrollings i, ∈uᵢP ↼ ∈'uᵢP. The simple rule is well-behaved, bounded, and local.

Figure 3 shows a loop unrolled and compacted using ∈. The only loop-carried dependency is between the first statement of consecutive iterations; after application of ∈ the iterations overlap, staggered by one node. The following lemma shows that detecting equivalent nodes is easy for the simple rule—nodes with the same statements from different iterations are equivalent.

Lemma 6.2. Let n and n' be nodes in ∈uᵢL, and let x be the occurrence of statement x in Lᵢ. Assume that n and n' are unaffected by larger unrollings and applications of
If there is a $k$ such that

$$x_j \in \text{op}(n) \iff x_{j+k} \in \text{op}(n')$$

and $n$ and $n'$ have the same dag structure, then $n$ and $n'$ are equivalent.

**Proof.** Assume $i$ is large enough that the successors of $n$ and $n'$ are unaffected by larger unrollings and applications of $\mathcal{C}$. The existence of $i$ is guaranteed by Lemma 5.4. Let $p$ and $p'$ be corresponding successors of $n$ and $n'$, respectively. Because statements from an iteration are adjacent and in the same order after the application of $\mathcal{C}$, it follows that $x$ is a statement of iteration $L_j$ in $n$ and $y$ is a statement in $L_j$ in $p$ if and only if $x$ is a statement of $L_{j+k}$ in $n'$ and $y$ is a statement of $L_{j+k}$ in $p'$. The only other possibility is that $p$ contains the first statement 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 blocked by the same statement from the previous iteration on a particular path. Thus, the first statement of iteration $L_j$ is in $p$ if and only if the first statement 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 the fact that $n$ and $n'$ have the same dag structure. Applying this argument inductively to $p$ and $p'$, it follows that $n$ and $n'$ are equivalent nodes.

6.3. An example

Figure 27 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 omitted the loop control code. 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.

```
\text{Test: If } A[i] = \text{key}
\begin{align*}
\text{t} & \quad t \\
\text{f} & \quad f
\end{align*}
\text{App: list := append(list,i)}
\text{i := i + 1}
```

Fig. 27. A simple loop $L$.

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

Existing restructuring transformations for multiprocessors do 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 [11]. 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; that is, iteration $i+1$ may begin after iteration $i$ has executed its first statement. The dynamic execution of this loop using doacross is shown in Fig. 28a. An equivalent static SPL schedule is shown in Fig. 28b.

We now show how perfect pipelining with the simple rule applies to this loop. In Fig. 29, the original loop is unrolled seven times. The statements have been

---

Fig. 28. Doacross applied to $L$. (a) Dynamic schedule; (b) static SPL program.

Fig. 29. $L$ unrolled seven times.
replaced by labels with subscripts indicating the increment to the induction variable. The result of applying the simple rule is shown in Fig. 30. 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 Fig. 30 are equivalent and the start node is equivalent to the first two nodes in the right column. Figure 31 shows the result of applying perfect pipelining—only the first two nodes remain. In this pipelined loop, three tests are executed at each step. If $T_j$ is the lowest numbered

\[
\begin{align*}
&\text{Fig. 30. } L \text{ unrolled seven times and compacted.}
\end{align*}
\]

\[
\begin{align*}
&\text{Fig. 31. The same loop after pipelining.}
\end{align*}
\]
test that evaluates to true, then the induction variable $i$ is incremented by $j$ and control passes to the node with the append statement. 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 TRACE, a commercial tightly coupled parallel architecture that supports multiway jumps [26]. In this example, the running time of perfect pipelining is dependent on the size (number of resources) of the machine as well as the original loop.

6.4. Comparison with doacross

As suggested in the previous section, loops transformed by doacross can be represented in our formalism. In fact, a restriction on perfect pipelining with the simple rule 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 for 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. (We assume that statements in the loop body are not reorganized.) 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 $D_{\text{synch}}$ be the compaction algorithm that implements doacross for synchronous multiprocessors. The restriction to the simple rule is made in move (see Fig. 32). 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 $D_{\text{synch}}L$, the first statement of iteration $i + 1$ overlaps iteration $i$ exactly where the communications are introduced by doacross.

The asynchronous case ($D_{\text{asynch}}$) 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.

```
procedure move(x, n, m)
    if x is an assignment
    then
        begin
            if $x \in op(s)$ for all $s \in succ(n)$ then
                $P := move-op(P, x, succ(m), m)$
            else $P := move-test(P, x, n, m)$;
        end
    Fig. 32. The move transformation for $D_{\text{synch}}$.
```
Again, the change is made in the move transformation. In \( D_{\text{asynch}} \), the first statement of iteration \( L_i \) may not move past a statement \( x \) in iteration \( L_{i-1} \) if any statement in \( L_i \) is dependent on \( x \). This is necessary because processors may run at arbitrary speeds; thus, it is not safe to start an iteration \( L_i \) until everything \( L_i \) may depend on has already been executed. The following theorem summarizes the relationship between the three transformations.

**Theorem 6.3.** For all loops \( L \), \( D_{\text{asynch}} L \preceq P D_{\text{asynch}} L \preceq P C u^\infty L \).

**Proof.** Follows immediately from previous discussion and the fact that \( D_{\text{asynch}} \) is a restriction of \( D_{\text{synch}} \), which itself is a restriction of perfect pipelining with the simple rule. \( \square \)

### 6.5. Efficiency

There are loops for which perfect pipelining with the simple rule requires time exponential in the size of the original loop. For example, consider a loop \( L \) with no loop-carried dependencies and a single test. If there are sufficient machine resources, then because there are no loop-carried dependencies the height of the program graph after compaction with the simple rule is a constant independent of the unrolling chosen. Now, \( u' L \) has \( 2^i \) paths through the loop body; preserving all paths in constant depth requires exponential code explosion. However, this can be detected after unrolling only once, because the iterations completely overlap after applying \( C \). In this case, the loop is completely vectorizable and generating good code is relatively easy.

Any compaction algorithm that is at least as powerful as the simple rule has worst-case exponential time complexity. In fact, every compaction-based parallelization system suffers from worst-case exponential performance [3, 12, 13, 24]. However, a good implementation can minimize the problem. Perfect pipelining often converges on many or all paths for unrollings much smaller than the worst-case bound; thus interleaving unrolling, compaction, and the test for equivalent nodes substantially improves the efficiency of the algorithm. Another possibility is to dynamically modify the compaction algorithm if code explosion becomes too severe. By sacrificing some parallelism, code explosion can be substantially reduced while still guaranteeing the convergence of perfect pipelining.

### 7. Conclusions

We have presented a transformational system for and a theory of program parallelization. Using the theory, we have shown that it is possible to compute limits of infinite sequences of the primitive transformations of our system. This limit-taking transformation allows our system to express the classical parallelization techniques for many machine models; in this paper, we have illustrated how a simple form of
vectorization and doacross can be expressed in our system. Thus, our transformational system and theory can be viewed a formal foundation for the area of parallelization.

Acknowledgment

Thanks go to Alex Nicolau for supervising this research, to Jennifer Widom for editing earlier versions of this paper, and to the anonymous referees for their careful reading and excellent suggestions.

References


