A Fine-Grain Parallelizing Compiler

Alexandru Nicolau

TR 86-792

December 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

A Fine-Grain Parallelizing Compiler

Alexandru Nicolau

Computer Science Department

Cornell University

Ithaca, NY 14853

Abstract

Percolation Scheduling (PS) is a new technique for compiling programs into parallel code. It attempts to overcome problems that limit the effectiveness and applicability of currently available techniques.

PS globally rearranges code past basic block boundaries. Its core is a small set of simple, primitive program transformations defined in terms of adjacent nodes in a program graph. These transformations constitute the lowest level in a system of transformations and guidance rules. Higher levels of this hierarchy control and enhance the applicability of the core transformations and enable us to exploit both fine grained and coarse parallelism.

Unlike other, more ad hoc approaches, PS is based on rigorous definitions of the computational model and of the core transformations. The correctness and termination of the transformations is proven here. The completeness of the transformations is also discussed. As a result our system implementation can proceed on a sound basis. In particular, PS enjoys greater adaptability and independence between the levels than would be possible otherwise. This has greatly facilitated the implementation of the first prototype version of our compiler, which is now complete.

This paper describes PS in detail. The correctness aspects as well as illustrations of the effectiveness of our techniques are presented. Architectures which may benefit from the use of PS are also discussed.

1 Introduction

The production of efficient, parallel code from high-level languages has been of major importance ever since parallel computation became practical. Efforts in this domain have spread in two complementary directions. The first seeks to exploit parallelism by designing new algorithms suited for parallel execution, often tailored to a particular type of architecture (e.g., systolic algorithms [16]). This approach is best suited for problems that can be naturally partitioned into a set of essentially identical subproblems that can run independently. The second approach involves compile-time transformation of a given program for efficient execution on a specific parallel architecture. While this usually does not change the asymptotic complexity of the original program, it can dramatically speed it up. Thus, the success of such compilation techniques is important both for automatically mapping parallel algorithms onto specific hardware, and for the efficient solution of problems that do not lend themselves well to partitioning. Even for problems for which parallel algorithms can be found, the algorithm design can be extremely tedious. For example it took years before researchers noticed a small transformation that makes LU decomposition suitable for vectorization, even though the problem was the subject of much attention.

Existing compilers for parallel machines do not provide the needed support. Code is still hand-optimized when speed is critical (e.g., [5]). While important advances in the parallelization of ordinary code (especially vectorization) have been achieved [10], [15], [2], there is still a lack of satisfactory tools to automatically extract fine and coarse grained parallelism in a unified fashion. The goal of Percolation Scheduling is to provide such tools. Thus PS can be thought of as complementing and/or enhancing previous approaches to parallelism extraction.

Percolation Scheduling (PS) is a new technique for compiling programs into parallel code. It developed from our experience with Trace Scheduling[8] in the ELI project at Yale [6], [18], [19]. Trace Scheduling was proposed as a global compaction technique for generating horizontal microcode [8]. In Trace Scheduling, directed acyclic paths (*traces*) through the original (sequential) code are selected and compacted as if they were *basic blocks* (sequence of program statements such that if the first statement gets executed, all the other get executed as well). Compensation code is inserted at jumps into and out of the trace to restore the correctness of the compacted code. Then a disjoint trace is selected and the process repeats. Traces are selected based on educated guesses of the execution frequency of the trace. This information is obtained through sample runs on "typical data" or is user supplied. The main limitations on the effectiveness of Trace Scheduling are its restriction to fine-grained parallelism, its inflexibility (the transformations are monolithic and dependent on the success of a single heuristic—trace selection) and its localized nature (only very fine-grained code is compacted, one trace at a time, and traces cannot be combined).

PS globally rearranges code past basic block boundaries in an attempt to gain parallelism. Its core is a small set of primitive program transformations defining the allowable motions of operations between adjacent nodes in a *parallel program graph*. A parallel program graph is a directed graph in which each node contains one or more operations that can be executed in parallel. The edges in the program graph determine the execution-paths in the program. The goal of PS is, then, to maximize parallelism by moving operations from node to node so as to maximize the number of parallel operations in the final graph. A precise definition of the execution semantics is found in section 3. PS core transformations are easy to understand and implement. Furthermore, they are atomic and can be combined with a variety of guidance rules to direct the optimization process. Above this core level are guidance rules and transformations which extend the applicability of the core transformations and exploit coarser parallelism.

Guided by the higher level rules and transformations, the core transformations operate uniformly on an entire program graph. They can also be applied to partially parallelized code. This allows modification of code produced by other types of compilers. In addition, these transformations are themselves highly parallel and could be run on a parallel machine, significantly reducing compilation time.

The following is an outline of the layers of the PS system and their function:

Core Level This level contains a set of four core transformations defined in terms of adjacent nodes in a program graph. Together these transformations perform all code motions in our system.

Support Level At this level we have analysis methods (e.g., Memory Disambiguation [18]) and standard optimizations (e.g., Dead-Code Removal). They provide accurate data-dependency information and thus enhance the applicability of the core transformations.

Guidance Level This level consists of rules that direct the application of the core transformations to achieve effective optimization of the code in acceptable time and space. This contrasts with Trace Scheduling where a single rule (for trace picking) is inseparable from the actual transformation mechanism. This limits Trace Scheduling and makes it too rigid for our goals.

Enabling Level This level consists of transformations that allow the core transformations to process arbitrary graphs and enables them to exploit coarser grained parallelism (e.g., within nested loops).

Meta Level This level consists of transformations that use the core transformations to exploit coarse parallelism (e.g., partial loop/module overlapping).

The separation of levels yields more general, cleaner transformations. It simplifies both the understanding and the implementation of a PS compiler. Furthermore, the applicability of the transformations is enhanced, and it is easier to experiment with various high level transformations without affecting the correctness of the compiler.

2 Architectures

The current implementation of the system is mainly geared towards statically-scheduled multiprocessors and very-long-instruction-word machines such as the FPS-264, FPS-164, Mars 432, the ELI-512, and horizontally microcoded engines. In this context, the need for tools of the type we propose has been widely recognized. Even traditional CISC machines with their large microcode programs that interpret higher-level machine instructions, could benefit from the use of our environment. Other architectures such as vertical lookahead (pipelined) engines could use the large numbers of sequential operations clustered together by our system to more efficiently fill pipelines.[1] Furthermore, since the compiler could do most of the work, there would be little need for hardware synchronization mechanisms and pipe flushing

[1] Current compile-time pipelining techniques used are mostly restricted to basic-block code motions. Since basic-blocks (i.e., sequential code between conditional-jumps) tend to be small in typical code, the room for rearrangement of code for smooth pipelining is likewise limited.

customarily used[2]. Hardware to handle multiple conditional jumps would improve the benefits of Percolation Scheduling for such machines. The design of such an architecture and its advantages are described in [13].

Also potentially suited to take advantage of percolation scheduling are data-flow architectures. Traditionally it has been claimed that such architectures will require very little compile-time effort. However, from a pragmatic point of view, this lack of compile-time effort will impose a very heavy burden in terms of communication and synchronization costs, and may lead to extremely inefficient use of memory and resources [11]. We believe that Percolation Scheduling could be put to good use in this context. PS can provide a *partial ordering* for the issuing of operations and a reasonable partitioning of the program and data between the various processors. This could significantly reduce runtime communication and synchronization needs as well as the lengths of queues of waiting operations. Furthermore the atomic nature of the core transformations and their independence makes PS attractive for both compilation for and running on data-flow machines.

3 The Model

Informally, we start with a program graph, in which each node represents one or more operations. The execution of such a program graph begins at a start node and proceeds one node at a time. The goal of PS then, is to transform the original program graph into another program graph which, while semantically equivalent to the original one, is more "parallel", i.e. the execution of the new graph traverses fewer nodes. This is achieved by the movement of operations from node to node, subject to data (and control) dependencies.

In this section we present the basic model of parallel computation that we use. A precise description of this model is necessary for the development and accurate statement of the transformations and for their proofs of correctness.

[2]Such hardware features can be cumbersome and expensive since they create hot spots which need to work faster than the rest of the hardware and operate asynchronously.

3.1 Preliminary Definitions and Notation

The following is a rigorous definition of a parallel program graph. Illustrations are found in figure 1.

A *node* in a parallel program graph is a three-tuple of the form $n \equiv < O_n,\ T_n,\ Next_n >$. O_n is a set of operations, $O_n \equiv \{o_1, ...\}$, such as assignments, additions, etc. Each operation o_i contains a set of *written* variables, a set of *read* variables and an *operator*. T_n is a set of conditional-jumps (cj's), i.e., $T_n \equiv \{cj_1, ...\}$. Each conditional-jump, cj_l, is a three-tuple $cj_l \equiv < test_l, Tnext_l, Fnext_l >$. The $test_l$ element of a cj is a test on a set of *read* variables (tests do not have side effects) and an optional *negation operator*. The fields $Tnext_l$ and $Fnext_l$ point to the control-flow followers (cj's in T_n, or other nodes) of cj_l. $Tnext_l$ and $Fnext_l$ correspond to the evaluation of the $test_l$ to $true$ or $false$ respectively. The last field of a node, $Next_n$, represents the unconditional continuation of that node and may point to another node m. If $T_n = \emptyset$, $Next_n$ determines the continuation of n; otherwise $Next_n$ is ignored and the continuation is determined by the evaluation of T_n. Members of O_n and T_n are referred to as *components* of n.

The cj's in T_n are constrained to implicitly define a binary tree inside node n as follows.[3] There exists exactly one tuple, cj_1, called the root, such that no cj_l in T_n contains a field $Fnext_l = cj_1$ or a field $Tnext_l = cj_1$ (i.e. no cj in n points to cj_1). For all other cj_l's in T_n there is at least one cj_j in T_n such that $Fnext_j = cj_l$ or $Tnext_j = cj_l$, but no cycles are allowed (by definition). Any fields of cj's in T_n not involved in forming the tree point to other program-graph nodes (i.e. point to outside n); these fields are *the continuations of* T_n. If the size of T_n is k, there are at most $k+1$ pointers from T_n to other nodes—when T_n is a balanced binary tree. The reasons for the structure of T_n and its semantics will become clear when we discuss execution semantics and the core transformations.

We will refer to the set of successors of n as continuations of n and denote this set by $*next_n$. This set is defined to be $*next_n = \{Next_n\}$ if $T_n = \emptyset$. Otherwise, $*next_n$ is the set of continuations of T_n.

Given a set of nodes N, the union of the $*next$ sets in N implicitly defines the edges

[3]Strictly speaking, some of the transformations may produce a DAG of conditional-jumps. However, for simplicity of exposition and implementation we can always transform such a DAG into a binary tree.

n1:<{X[a]: = 1; a: = a + 1; tst: = a ` 30; tst-xa: = a< 10}, nil,n2>

n2:<nil,{cj1: if tst-xa then n3 else cj2: if tst then nil else n1},nil>

n3:<{X[a]: = 10},{if tst then nil else n1}, nil>

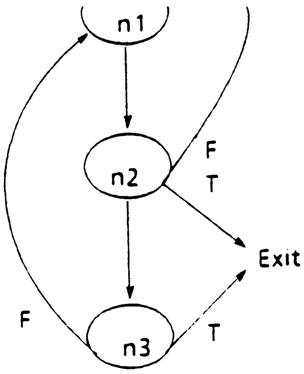


Figure 1: Sample program graph. (a) Program Graph. (b) Graphic representation.

of a *(parallel) program graph*, $P = (N, E)$. Each element n in the set $*next_m$ of node m defines an edge $e = (m, n)$ in E, where m is the *tail* and n the *head* of the edge. A *path* between two nodes n and v is the sequence of nodes and intervening edges explicitly denoted by $n, (n, l), l, (l, m), ..., (u, v), v$. If unambiguous, the shorthand notation $n \rightarrow v$ will be used. A simple parallel program graph and its graphic representation are shown in figure 1.

We are now in a position to describe the execution semantics of a program graph. This is needed to understand the rationale for the dependency definitions and the primitive transformations.

3.2 Execution Semantics

A program graph, P, starts execution at a start-node (a node with no incoming edges) and proceeds one node at a time, until an $\emptyset$-continuation is reached. The execution of a node, n, consists of the *evaluation* of all $o \in O_n$ and *tests* $\in T_n$. The order of evaluation of the components of n is immaterial as long as all reads of a variable in n precede the writing of that same variable in n. This may appear as a strange constraint. However it is consistent with the way standard memories handle reads and writes of the same location in the same cycle. The model simply reflects this feature, allowing in the same node the simultaneous execution of reads and a single write of the same location.

The next node to be executed, denoted $continuation(n)$, is chosen to be the node obtained by the evaluation of the *tests* in T_n and the resulting decisions ($false \Rightarrow Fnext$, $true \Rightarrow Tnext$). If $T_n = \emptyset$, then $continuation(n) = Next_n$.

Some explanation of the semantics of conditional-jump execution is in order. A conditional-jump can (by definition) have 2 branches, which in our case are represented by $Tnext$ and $Fnext$. When the n_i only contains one cj (i.e. a simple conditional-jump), the above semantics fit normal program execution semantics on a standard von-Neuman machine. However, n independent cj's in a node would in general have $2n$ possible continuations, whereas our model only allows for $n+1$ (see definition of T_n). To fully utilize the evaluation of n independent conditional jumps in parallel, execution would have to be able to continue execution on the resulting n mutually independent paths in parallel, or alternatively would require a selection between 2^n paths. However cj's at this level of granularity are not independent. If, in particular, they all share a common path, (e.g. like in a case statement), a simple and effective scheme can be devised for their practical parallel resolution. Such a scheme was sketched in [7] and a more general design, allowing DAG's of conditional-jumps, was described in [13]. The key idea is that n cj's sharing a path or forming a decision-DAG have at most $n+1$ *mutually exclusive branches*. Since this sort of parallel evaluation is likely to be useful and is quite cheap to implement in hardware (see [13]), we have biased our model and transformations towards such parallel evaluation. This bias can be easily lifted. In fact, the meta-level in our hierarchy will extend the notion of a node to deal with independent cj's at granularity levels where this may be useful.

3.3 Dependencies

We will now define dependencies between nodes and between components of nodes. These dependencies are used to restrict the core transformations to semantics-preserving code motions.

A *dependency* between two adjacent nodes, m and n, is said to exist when any one of the following holds:

D_{wbr}. *Write-Before-Read* : A component of m writes a variable read by a component of n. This is a standard data-dependency.

D_{wl}. *Write-Live* : There exists $(m, x) \in E$ such that variables *live-on* (m, x) are written by a component of n. A variable is said to be *live-on* (m, x) if there exists a path through (m, x) on which the variable is used (read) before being written. For a rigorous definition of *live* and *reaching* see [1].)

D_{war}. *Write-After-Read* : There exists components c and c' in the same node n such that c reads variables written by c'. For uniformity, this is also defined to be a dependency between

n and any adjacent node m. [4]

We will denote dependencies between two nodes by $m \prec n$, or $m \overset{x}{\prec} n$ when we wish to emphasize that a particular dependency, D_x, between m and n. The components creating the dependency between the nodes are also said to be dependent. Since the distinction between component and node dependencies is clear in context, we will use the same notation for both.

4 Core Transformations

The four primitive transformations of this level are the core of PS. They operate on adjacent nodes in a program graph. Repeatedly applying the transformations allows components to "percolate" (i.e. move forward) from the various parts of the program graph towards the start node, hence the name Percolation Scheduling.

A node in the program graph can be removed by the *Delete* transformation when the node has no components. The *Move-op* and *Move-cj* transformations move simple operations and conditional-jumps respectively, between two adjacent nodes. The *Unify* transformation unifies identical operations occurring in multiple neighbors of a node. The details of the transformations deal with maintaining the integrity of all paths affected by the transformations. A rigorous definition of the transformations is given bellow. The changes to nodes in N are given explicitly. The changes to the set of edges E are implied by the changes to N.

Def: A **copy**, n', of a node n is a new node with the same components as n. All continuations not pointing to n are the same as in n, while continuations of n' pointing to n are changed to point to n'.

[4]The dependency is really a non-strict dependency between c and c', forcing c not to precede c' on any path through m and n. In the original program graph c must have occurred after c'. Otherwise, if c had occurred before c', dependency D_{wtr} would have prevented c' from being in the same node as c. Since we are interested in code movement, the definition emphasizes that c cannot move to m.

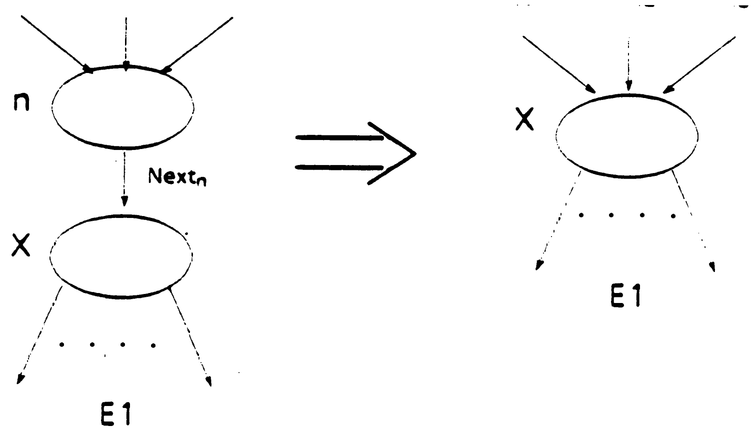


Figure 2: Delete Transformation

4.1 Delete Transformation

Nodes without any components may occur as a result of the other transformations or as part of the original program graph. Since they do not affect the execution semantics of the program in any way, such nodes may be deleted, provided the continuations of their predecessors are set accordingly.

Preconditions:

(a) A node $n = < \{\}, [], Next_n >$. That is, n has no components.

The Transformation:

1. For each $x \in N$, each element e of $*next_x$ that points to n is changed to point to $Next_n$.

2. $N := N - \{n\}$.

An illustration is given in figure 2.

4.2 Move-op Transformation

This transformation moves an operation component from node n to node m, through edge (m, n), provided no dependency exists between m and the component being moved. In performing the movement, care must be taken not to affect paths passing only through n but not through m.

Preconditions:

(a) An edge (m, n) exists.

(b) Node n has the form $< \{o, ...\}, [...], Next_n >$. That is, O_n has at least one component.

(c) $m \not\delta o$.

(d) $n \neq m$.

The Transformation:

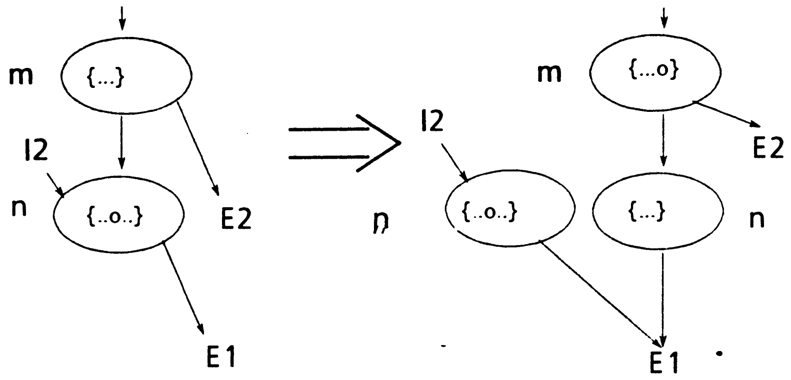


Figure 3: Move-op Transformation

1. $n' := copy(n)$.

2. For each $x \in N$ such that $x \neq m$ and $x \neq n$, change each element e of $*next_x$ that points to n so it points to n'.

3. $O_m := O_m \cup \{o\}$; $O_n := O_n - \{o\}$.

4. $N := N \cup \{n'\}$.

An illustration is given in figure 3.

4.3 Move-cj Transformation

This transformation moves a conditional-jump component from node n to node m through an edge (m, n), provided that no dependency exists between m and the component being moved. In performing the movement, care must be taken not to affect paths passing only through n but not through m. In addition, since the component may come from an arbitrary spot in the binary tree T_n, n will be split in order to maintain the execution semantics of paths through (m, n).

Preconditions:

(a) An edge (m, n) exists.

(b) n has the form $< \{...\}, [...c...], Next_n >$ That is, T_n has at least one component.

(c) $m \not\to c$.

(d) $m \neq n$.

If c has a parent in the tree T_n, call that parent p. Assume that $Tnext_p = c$ (if $Fnext_p = c$, we can always negate $test_p$ and swap). Let S_{Fnext} be the set of nodes in the subtree of T_n pointed to by $Fnext_c$ and S_{Tnext} the set of nodes in the subtree of T_n pointed to by $Tnext_c$. If no such subtree(s) exist (i.e. if c points to other nodes), let $S_{Tnext} = S_{Fnext} = \emptyset$.

The Transformation:

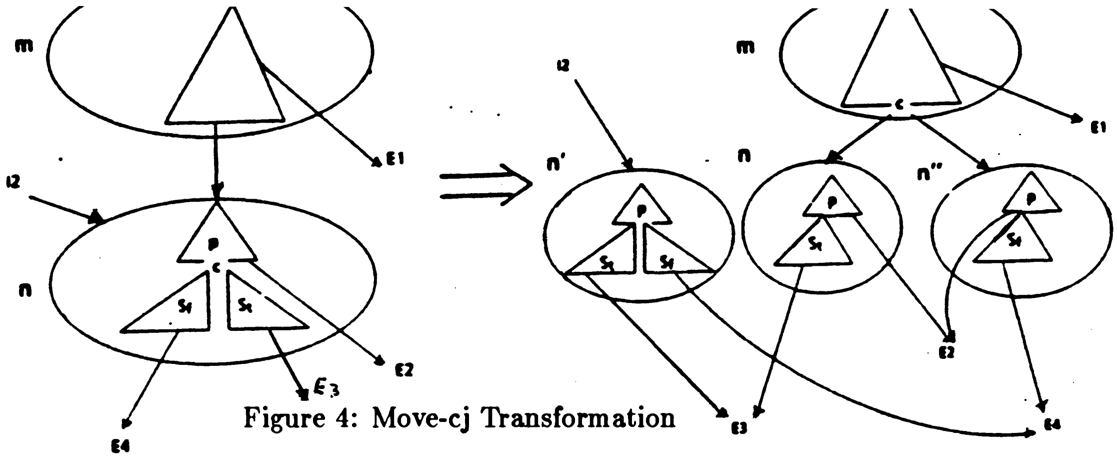


Figure 4: Move-cj Transformation

1. $n' := copy(n)$.

2. For each node $x \in N$ such that $m \neq x$ change each element e of $*next_x$ that points to n to point to n'.

3. Change n to be $n =< O_n, T_n - \{c\} - S_{Tnext}, Fnext_c >$; If p exists, then set $Tnext_p = Fnext_c$.

4. Let n'', a new node, be $n'' =< O_n, T_n - \{c\} - S_{Fnext}, Tnext_c >$; If p exists, then set $Tnext_p = Tnext_c$.

5. $T_m := T_m \cup \{c\}$; $Next_m := \emptyset$. Any $cj \in T_m$ with a field pointing to n is changed to point to c.

6. $Tnext_c := n''$; $Fnext_c := n$.

7. $N := N \cup \{n', n''\}$.

An illustration is given in figure 4.

4.4 Unification Transformation

This transformation moves a unique copy of identical operations from a set of nodes $S = \{n_0, n_1, n_2, ...\}$, to a predecessor node m. This is done only when no dependency exists between m and the component being moved and when paths (m, n_i) exist for all nodes in S. In performing the code motion, care must be taken not to affect paths going only through the n_i's but not through m.

Preconditions:

(a) There exists an edge $(m, n_0) \in E$.

(b) For all $n_i \in S$, there exists a path from m to n_i.[5]

(c) The chosen operation $o \in O_{n_0}$, has as its unique dependency $m \overset{wl}{\prec} o$.

[5]For efficiency we may wish to restrict the transformation to cases where all $n_i \in S$ are adjacent to m.

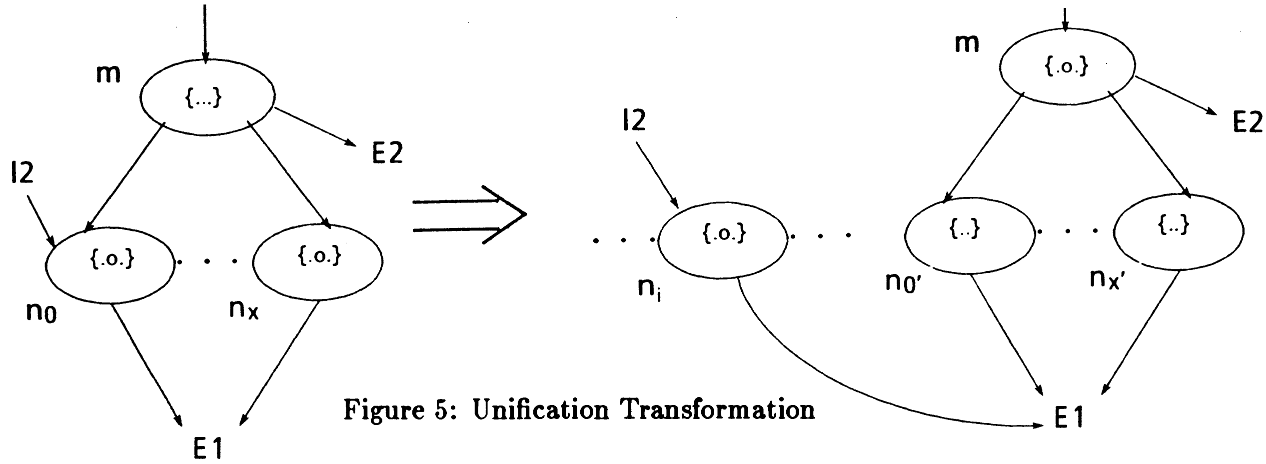


Figure 5: Unification Transformation

(d) All the components on paths through m that read variables written by o are identical copies of o, denoted o'.

The Transformation:

1. For $o \in n_0$: apply the Move-op transformation, moving o from n_0 to m.

2. For other $n_i \in S$: remove o' as in the Move-op transformation, but do not place it in m (just remove it).

An illustration is given in figure 5.

5 The Correctness of the Core Transformations

There are three separate issues that we have to address. First, we will discuss *completeness*. That is, given our model, we want to know whether our core transformations can achieve all possible code motions. Next we will show that the core transformations of Percolation Scheduling are *semantically correct*. That is, given the same input, the original and transformed program graphs, executed according to our model's execution semantics, will produce the same output. Finally we will show that the process of applying the transformations *terminates*. That is, given any input program graph, after a finite number of applications of the transformations we will obtain a program graph to which no more core transformations can apply.

5.1 Completeness

A truly complete system of parallelizing transformations, is in general, unfeasible. However, by studying the relative completeness of our transformations we hope to discover new transformations and be able to precisely describe the power of our system. The work we've done to date in this area has lead to the discovery of transformations that we haven't originally envisioned (e.g., unify, dead-path elimination) and that make our system strictly more powerful.

Informally, the core transformations we have defined are *weakly complete*. That is, given the ability to backtrack they can achieve all possible code motions based on *local* information—i.e., information available at the adjacent nodes or as a result of flow-analysis. We assume that backtracking or backward movement from the original program graph are not allowed. Enhancement transformations such as loop invariant and dead code removal, code hoisting, etc. are orthogonal to this discussion. They will, in fact, be performed by the higher levels of PS, as will other traditional optimizations.

In this context, backward-movement (away from the start node) transformations cannot achieve better results than our core transformations. They are either incorrect (if they move an operation which ought to be on a path out of that path, violating the execution semantics of that path[6]), or are undoing a forward transformation (i.e. backtracking).[7] Thus we only need consider forward-movement transformations.

The Delete transformation just garbage-collects. it is irrelevant in this context. It could be viewed as the cleanup phase of the other core transformations.

The Move-op and Unify transformations cover all local code-motions of operations (subject to dependencies).

All local code-motions of conditional-jumps (subject to dependencies) are covered by the Move-cj transformation.

The effect of moving compound nodes (as a whole) can be achieved by the composition of core node motions.

We could strengthen our completeness arguments by allowing other transformations (e.g., allowing backward movement beyond the original program). However, allowing more transformations is not necessarily desirable, since it may destroy the *forward progress* properties of the core transformations. Forward progress is important both for ensuring termination and

[6]Assuming that the operation was performing a useful function at that point; if it didn't, it would have been removed by dead code removal.

[7]Backtracking is expensive and a practical system will most likely restrict it, sacrificing completeness. This argument just ensures that backward movement does not produce desirable effects other than nondeterministic choice.

for any practical implementation. We will reexamine the completeness issue once we gain some experience with our prototype implementation and are thus better able to judge the interaction between the higher levels and the core transformations.

This informal discussion is not meant to establish full completeness for the core transformations. It would be naive to expect such results for the core transformations. The discussion is just meant to give some feeling for what the core transformations can and cannot do, and give us a basis for deciding on what other transformations we may want at the higher levels.

5.2 Semantic Correctness

We will now show that the core transformations preserve semantic correctness by proving that the application of any of the transformations does not affect the semantic correctness of any path through the program graph. In talking about a path, we implicitly assume that whenever a path is taken, the tests in cj's along that path evaluate to result in execution along that path. This is a natural assumption, since otherwise we would be dealing with a different path.

Claim 1: The Delete transformation preserves semantic correctness.

Proof:

No components are removed or introduced on any path by this transformation, and the paths through the removed node are not affected (i.e. they are made to point to the continuation of the deleted node). The deleted node having no function, its removal does not affect the execution semantics. □

Claim 2: The Move-op transformation preserves semantic correctness.

Proof:

1. Only paths through m or n are affected.

2. There are at most two types of entry points into the affected part of the graph: one into m ($I1$) and one into n ($I2$). There are two types of exits: one from n ($E1$ in T_n or $Next_n$) and one from m—excluding the edge (m, n)—($E2$), see figure 3. This yields the following possible paths through the affected nodes:

$I1 \rightarrow E1$**:** Moving o from n to m cannot violate dependencies, (i.e. cannot affect the evaluation of components in n or m along this path) and no new components are added to this path by the transformation. Changes to continuations of nodes in N that pointed to n other than (m,n) are irrelevant for this path since these continuations define different paths, and since their evaluation could not have changed (all operations and dependencies being preserved). Thus the execution semantics of this path cannot be affected.

$I1 \rightarrow E2$**:** As above, all components on this path and the dependencies among them are preserved by the transformation, and cj's on this path are unaffected. While o is foreign to this path (i.e. wasn't there before the move), it cannot affect the execution semantics of the path (or any paths containing it) since otherwise dependency D_{wl} would have prevented the move.

$I2 \rightarrow E1$**:** The execution semantics of this path are completely unaffected, as n' is an identical copy of the original n. □

Claim 3: The Move-cj transformation preserves semantic correctness.

Proof:

1. Only paths through m or n are affected.

2. There are at most two types of entry points into the affected part of the graph: one into m ($I1$) and one into n ($I2$). There are four types of exits: one from m ($E1$), not across the edge (m,n) along which the transformation occurs, and three from n: $E2$—exits for which the decision does not go through the moved component c; $E3$—exits for which the decision goes through $Tnext_c$; and $E4$—exits for which the decision goes through $Fnext_c$. See figure 4 for an illustration. This yields the following types of paths:

$I2 \rightarrow E2$ **or** $E3, E4$**:** n' is an identical copy of the original n so the transformation preserves the execution semantics of these paths.

$I1 \rightarrow E1$**:** For such a path to exist we must have $T_m \neq \emptyset$. After the transformation, c is a member of T_m, but its position with respect to the rest of the cj's in T_m is unchanged (by construction) and dependencies are preserved for all operations in m. Furthermore c is side-effect free and thus won't affect any operations on paths to $E1$. So if the evaluation of the original m leads to $E1$ being reached originally, c is irrelevant and thus $E1$ will also be reached in the transformed graph. That is, the execution semantics are preserved for this path.

$I1 \rightarrow E2$**:** Since cj's that may lead to $E2$ being reached are not part of either S_{Tnext} or

S_{Fnext}, they appear (unchanged) in both T_n and $T_{n''}$ after the transformation. Since c is now part of m, and no other operations have moved and no dependencies are violated, c will be reached while evaluating m. While c may choose either n or n'', both these nodes are semantically equivalent with respect to $E2$, since they contain all the simple operations originally in n and all the cj's that may lead to an exit $E2$ being taken. Thus all paths from $I2$ to $E2$ have the same execution semantics as the original path. Note that this still holds if the original $T_m = \emptyset$ (c becomes the root and sole element of T_m).

$I1 \rightarrow E3$: As for $E2$, the evaluation of m causes c to be reached. Everything being equal (dependencies and operations are preserved, and c being added to m does not affect the computation), c's decision will be the same as originally. Assuming $E3$ was reached before, c will pick $Tnext = n''$ as the continuation. In n'' all components of O_n are present and all components of T_n, except c and its $Fnext_c$ subtree. Everything being equal, the point where c was located in the original T_n is reached while evaluating $T_{n''}$. Since at this point in n'' the root of S_{Tnext} is found, the continuation is picked by S_{Tnext}'s evaluation. That is, some $E3$ is reached. While S_{Fnext} is not in n'', it is irrelevant if $test_c = true$. Thus the new path is semantically equivalent to the original path to some $E3$. Note that the above still holds if c is the only cj in n (then $T_n = \emptyset$ and $Next_{n''} = E3$), when c is the root of the original T_n (then $T_{n''} = S_{Tnext}$) and when $S_{Tnext} = \emptyset$ (then $Tnext_p = E3$ in n'').

$I1 \rightarrow E4$: Same argument as for $I1 \rightarrow E3$, with n replacing n'', $E4$ replacing $E3$, $Fnext$ replacing $Tnext$ and $false$ replacing $true$. □

Claim 4: The Unification transformation preserves semantic correctness.

Proof:

1. Only paths through nodes $m, n_0, \ldots n_x$ (see figure 5) can be affected. There are two types of entry points: one into m ($I1$) and one into $n_0, \ldots n_x$ (I_2). There are only two types of exits: one from $n_0, .., n_x$ ($E1$) and one from m ($E2$). See figure 5) for an illustration. This yields the following types of paths:

$I1 \rightarrow E1$: Completely subsummed by Claim 2 above, since each such path (through each of $n_0, \ldots, n_x$) is equivalent to path $I1 \rightarrow E1$ in Claim 2.

$I1 \rightarrow E2$: Completely subsummed by Claim 2 above, since each such path is equivalent to path $I1 \rightarrow E2$ in Claim 2.

$I2 \rightarrow E1$**:** Completely subsummed by Claim 2 above, since each such path is equivalent to path $I2 \rightarrow E1$ in Claim 2. □

5.3 Termination

In practice we will guide the application of the transformations described in the previous section by some guidance rules (e.g. move an operation as high as it will go before moving another one). However we do not want to restrict our arguments to a particular case. We will show termination in the most general case, i.e., when the transformations can occur in any order.

We only need consider Directed Acyclic Graphs as input program graphs. Without some global information about loops to provide a sense of direction, there is no *reason* why the core transformations should stop applying if there is some cycle in the graph. To deal with cyclic graphs we simply chose points at which to introduce start and exit nodes, so we only have directed acyclic graphs at the core transformations level. The point at which cycles are broken is arbitrary for the purpose of the proof, but may be important in a practical system. Methods for cycle breaking are provided by the higher levels of PS and are discussed in the next section.

Given a DAG, the *depth* of a component (or of the whole node) is the maximal number of nodes between that node and the start of any path reaching that node—i.e. the "potential movement" of components of that node. Two paths in the DAG are said to be *distinct* if they have at least one distinct edge. A path is *maximal* when it is not a subpath of some other path. The *length* of a path is the number of nodes along that path.

We show that the application of any core transformation to an adjacent pair of nodes m and n decreases the depth of at least one component c on some path p. Any components introduced by the transformation on existing paths have depth less than the original depth of c. Furthermore, a transformation may create at most two paths for each original p it replaces, each with fewer components of the same depth as the original component c, and with the same components as path p for depths greater than that of the original c. We can show that these facts are enough to ensure termination, because the motion of any component is bound by its original depth.

Claim 5:

Given any DAG, any arbitrary sequence of applications of core transformations must eventually terminate.

Proof:

For each *distinct* maximal path p, consider the tuple

$$< N_1^p, N_2^p, ..., N_k^p > .$$

N_i^p is the number of components on path p with depth equal to i. Now let us examine the effect of each transformation on these tuples.

Delete Transformation: This transformation only removes empty nodes. While this does not reduce the number of components on a path, it may reduce the depths of components below the node being removed. Even when this doesn't happen (e.g., when the node removed was the last one on a path), the node is removed from the graph, so it cannot be removed again. Thus at worst this transformation has no effect on any tuple $< N_1^p, ...N_x^p.... >$, but decreases the length of the associated paths by one. Since the transformation cannot create new empty nodes by itself, it cannot be executed unless some other transformation creates an empty node. If the other transformations cannot apply (i.e. have "terminated"), the Delete transformation must terminate as well after deleting all empty nodes.

Move-op Transformation: Given a transformation on edge (m, n), then with respect to paths containing edges $(*, n)$, where $*$ is not m, two possibilities arise. It may be that no change occurs, even in the event of copying, because the components of n were already counted as part of these paths when assembling the tuples, and the depth of n was determined by one of these paths. Alternatively, the depth of n was not determined by these paths, and then copying will only reduce its depth on these paths by decoupling the paths from those imposing the current depth. Thus if we had a tuple $< N_1^p, ...N_x^p.... >$, it either stays unchanged, or we get $< N_1^p, .., N_{x'}^p +, ...N_x^p - .... >$, where x is the depth of n before the transformation, p is any path through $(*, n)$, x' is the depth of n' on path p after the transformation, and $+/-$ stand for increment/decrement.

With respect to any paths containing m and n, there is one less component of depth x and one more component of depth at most $x - 1$. If the depth was not determined by a path through (m, n) the depth of the moved component could actually be smaller than $x - 1$. So, in the worst possible case we will get for any such path q, a tuple $< N_1^q, ..., N_{x-1}^q+, N_x^q-, ... >$. Paths through m but not through n have at worst gained a component of depth at most $x-1$ (if the depth of n was established through m, and even less otherwise). Thus for any such path r, the net effect is at worst $< N_1^r, ..., N_{x-1}^r+, ... >$.

Note that the number of paths is not increased by this transformation, as paths resulting from copying were already distinct and thus counted separately. Furthermore, no path-length increases as a result of this transformation. □

Move-cj Transformation:

For paths containing $(*, n)$ where $* \neq m$, the argument is the same as for Move-op.

For paths only containing m but not n, the argument is again the same as for Move-op.

For paths through (m, n), moving c causes new paths to be created. If there were d paths containing (m, n), there will now be at least $d+1$ and at most $2(d-1)$ paths as a result of having n and n'' on the branches of c. More precisely, each path q through (m, n) yields as a result of the transformation at most two paths q' and q'', with tuples $< N_1^{q'}, ..., N_{x-1}^{q'}+, N_x^{q'}-, ... >$ and $< N_1^{q''}, ..., N_{x-1}^{q''}+, N_x^{q''}-, ... >$. Paths q' and q'' differ in only one edge ((m, n), (m, n'')) and for all i, we have $N_i^q \geq N_i^{q'}$ and $N_i^q \geq N_i^{q''}$. (Duplicate paths are only created to ensure the semantic correctness of paths $I1 \rightarrow E2$—see figure 4.) □

Unification Transformation:

Same as for Move-op: at least one path contains less components of depth x. □

Thus the net effect of Move-op, Move-cj and Unify transformations is to decrease N_k for some k in some tuple, while no other tuples have N_k increased and all new tuples have smaller N_k's than the tuples they replace. In particular, to increase the number of components of depth $x-1$ on any path as a result of these transformations, the number of components of depth x on the same or other paths must decrease. None of the transformations apply unless the depth of the moving component is greater than or equal to 2. It follows that even though the number of components and paths may grow, the process will terminate if the length of all paths is bounded by some k. Since we have seen that no transformation can increase the length of a path, such a bound k exists: it is the maximal length of a path through the program graph before any transformations are applied. If all tuples have at most k elements, once the k^{th} element in a tuple starts decreasing it will eventually have to become 0 for all paths, since it cannot be increased again. If the x^{th} rather than the k^{th} element is decreased ($x < k$), the effect is the same except that x will be able to increase again, as a result of decreasing elements $> x$. The fact that a bound k exists ensures that this cannot go on forever. Since the Delete transformation can neither create new paths nor new nodes nor increase any element of a tuple, it cannot affect the termination of the other transformations, and thus its own starvation is implied by that of the other transformations. Thus overall termination is assured. □

5.4 Inverse Transformations

Inverses of the core transformations can be defined. The formulation is straightforward for the delete, move-op, and unification transformations. The conditions under which a conditional can be moved from a node N to a successor node M are somewhat more complex and are not presented in this paper.

The inverse transformations are used to undo previous transformations. This is sometimes desirable because an operation i can "percolate" to a node where it prevents another operation j from moving. If it would be more advantageous to move j rather than i, then i must be moved by a sequence of inverse transformations to a point where it no longer blocks j.

6 Beyond the Core Level

A realistic compiler that exploits parallelism of the scope we have proposed has to cope with efficiency issues as well as provide a means of exploiting more global forms of parallelism than the core transformations can achieve by themselves. This is the function of the higher layers of PS.

6.1 Support Layer

This layer identifies data dependencies between the components of nodes in the program graph. The methods outlined below enhance the accuracy of data-dependency analysis and as a result widen the applicability of the core transformations.

Disambiguation and Flow Analysis: Disambiguation tries to predict, to the extent achievable at compile time, whether two indirect memory references will conflict. It relies heavily on flow information. Runtime Disambiguation (RTD), a mechanism developed for Trace Scheduling may be helpful here as well. Techniques for straight disambiguation and RTD, as well as for enhanced flow analysis are described in [18].

Renaming: This is a basic technique that eliminates multiple assignments to the same variable. It is used in most parallelism exploitation methods. It relies on flow information. It could be done once when compilation begins, but could be useful if done incrementally during the transformation process as well.

Traditional Optimizations: Standard optimizations such as Common Subexpression Elimination, Dead-Code removal, and Loop Invariant elimination are also useful during the transformation process. For example, dead (redundant) code may be created by the transformations. It may be possible to tune the core transformations not to generate such redundant code. Such tuning is likely to be more efficient than repeating these optimizations globally.

An interesting variation of dead-code removal is *dead path elimination*. Moving a conditional-jump can produce a program graph with some paths that can never be taken. In Fig. 6, moving conditional i from B to A results in the copy of i in A dominating the copies of i in B_t and B_f. Because the truth value of i's boolean condition cannot change between A and B, the copies of i in B_t and B_f are redundant and can be deleted from these nodes. Not all cases where dead paths are created can be viewed so locally; it is easy to construct examples in which the movement of a conditional makes another conditional in a distant node redundant.

The technique we use for detecting redundant conditionals is essentially an application of range analysis [9], a type of data-flow analysis. Two sets are associated with each node: FC-LIVE and FC-REACHING (FC stands for flow-of-control). Using standard data-flow terminology, FC-LIVE(X) is the set of conditionals that are live at X. Conditional i is live at X if there is another node Y such that i is an operation in Y and there is a path from X to Y along which no variable read by i is defined. Similarly, FC-REACHING(X) contains pairs (i,T) and (j,F), where (i,T) (resp. (j,F)) reaches node X if each path to X evaluates i (j), the result is true (false), and no subsequent node on the path before X defines any variable

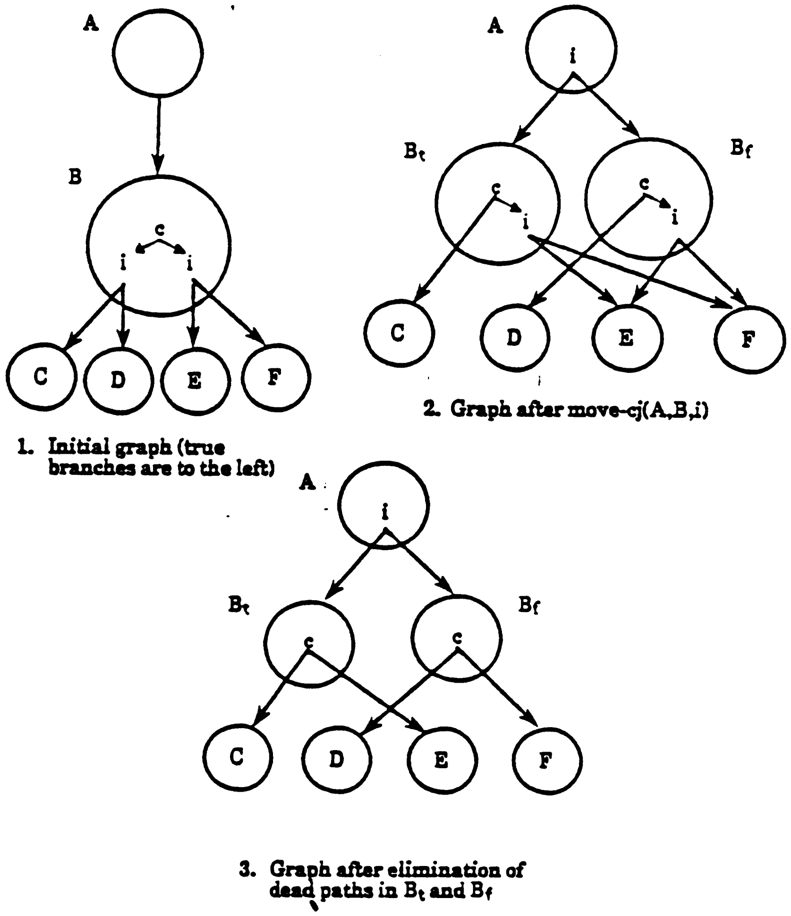


Figure 6: Removing a redundant test.

read by i (j). The data-flow equations for FC-LIVE and FC-REACHING are similar to the standard equations for live variables and copy propagation respectively.

The detection and elimination of dead paths is done as follows. Assume a conditional i is moved from node N to node M. If (i,T) or (i,F) reaches M, then i is deleted from M and the appropriate edge discarded. Otherwise, if there is any node X reachable from M such that i is an operation in X and (i,T) or (i,F) reaches X, then i may be deleted from X. The same procedure is also applied to the copy of N (if there is one). Note that eliminating an edge may leave a node in the graph (other than the start node) with no predecessors. This node, and possibly some of its successors, is unreachable and may be deleted.

The method described here can be made considerably more efficient. In particular, FC-LIVE information can be used to determine whether there is any node X reachable from a node M that uses a conditional i. A further improvement would be to maintain a set for each conditional i consisting of all the nodes containing i.

6.2 Guidance Layer

This layer guides the core transformations to improve efficiency (i.e. avoid backtracking). The following is a sampling of heuristics that, based on our experience to date, could be useful. We expect this level as well as the higher ones to be fine-tuned as a result of experimentation with our prototype.

Node Movement Precedence: Move-op transformations are given precedence over Move-cj and Unify transformations. Once a component has moved it is given precedence over components that have not moved yet. This is similar to flow information propagation, and may converge fastest if we proceed in reverse depth first order. The rationale for this rule is to avoid unnecessary code duplication and redundant movement of the same operations past each other.

Node Probability: Moving nodes with highest probability of execution is given priority. This is similar to trace scheduling, but much more general. It makes PS applicable even in situations where the notion of a trace is not well-defined. This rule applies to many conditional jumps, particularly in Quantized Loops (see below). However the predictability of conditionals, while desirable, is by no means critical to Percolation Scheduling.

Unpredictable Conditionals: On branches of conditionals that appear during compile-time analysis to have equal probability of execution, nodes executed on both branches are given priority. Subject to the availability of resources, operations in nodes with the longest chains of dependents are allowed to move first. Ties are broken by alternating between the branches. This will always guarantee some speedup, unlike previous methods where the wrong choice may result in a slower program than the original one.

Compound Node: Whole-node movement is given priority over movement of its components. The idea is to avoid fragmentation, and possibly save some dependency testing time. This is not necessarily well-defined for arbitrary compound nodes, but is obvious for nodes containing no conditional jumps.

Probabilistic Transformations: The primitive transformations are randomly applied (but biased by different probabilistic weighting). This would be close to the Monte Carlo approach [12], and is probably impractical. However, it has two significant advantages over the original Monte Carlo approach that would improve its performance: guaranteed correctness (no need to check or retract transformations) and guaranteed forward progress. This approach can be further enhanced by taking into account the probabilities of execution of the operations themselves based on a combination of flow analysis and their importance to computations underway (i.e. length of dependency chains originating at them).

Human Help: The user could guide the application of the primitive transformations. This need not be tedious; the system would perform the necessary transformations and bookkeeping and would maintain the correctness of the program. In fact this approach together with a small subset of high level transformations (for obvious situations and cleanup) could provide the core of a support environment for parallel program development.

6.2.1 Scheduling Transformations

In our compiler, we have built in several aggregate transformations that actually compact the code using built in heuristics. Because they are built on top of PS, however, these transformations are modular, flexible, and easy to change.

A simple transformation that we have implemented in the system, called *move-path*, takes as its arguments a source node A, an operation i, and a destination node B. *Move-path* then generates a sequence of core transformations that will move operation i from A to B if semantic correctness is not violated. If a potential data dependency violation is discovered, the operation is moved as far as possible and the conflict is reported to the user.

A more powerful transformation is *migrate*. *Migrate* moves an operation as far "up" in the graph as dependencies allow. This includes moving any copies of the operation that are created in the process. We wish, naturally, to perform a unification whenever possible. Opportunities for unification arise when an operation is copied on different paths–perhaps several times–and then at least some of the copies can be moved to the point where the paths rejoin. For example, assume that statements i and j in Fig. 7 can move on all paths above node A. It would be unfortunate to miss a unification here. In the case of statement i unnecessary copies of the operation would be left in the program, wasting space and consuming resources in the final code. Operation j cannot move into nodes A and C unless unifications are performed because of data dependency conflicts with other copies of j.

As an example of our transformations we develop *migrate* in detail. Let i denote the operation we wish to move and let C(i,t) denote the set of all copies of i in the program graph at time t. We define the function node(j) to be the node containing operation j. Finally, we assume for the moment that the program graph is acyclic.

It is easy to show that any algorithm which satisfies the following two conditions will perform all possible unifications:

1. Let t be the time at which the algorithm terminates. Then there is no j in C(i,t) such that j can be moved from node(j) to any predecessor of node(j).

2. Let reach(Y) denote the graph of all nodes reachable from a node Y. If j is moved from node(j) to some predecessor X of node(j) at time t and X has multiple successors,

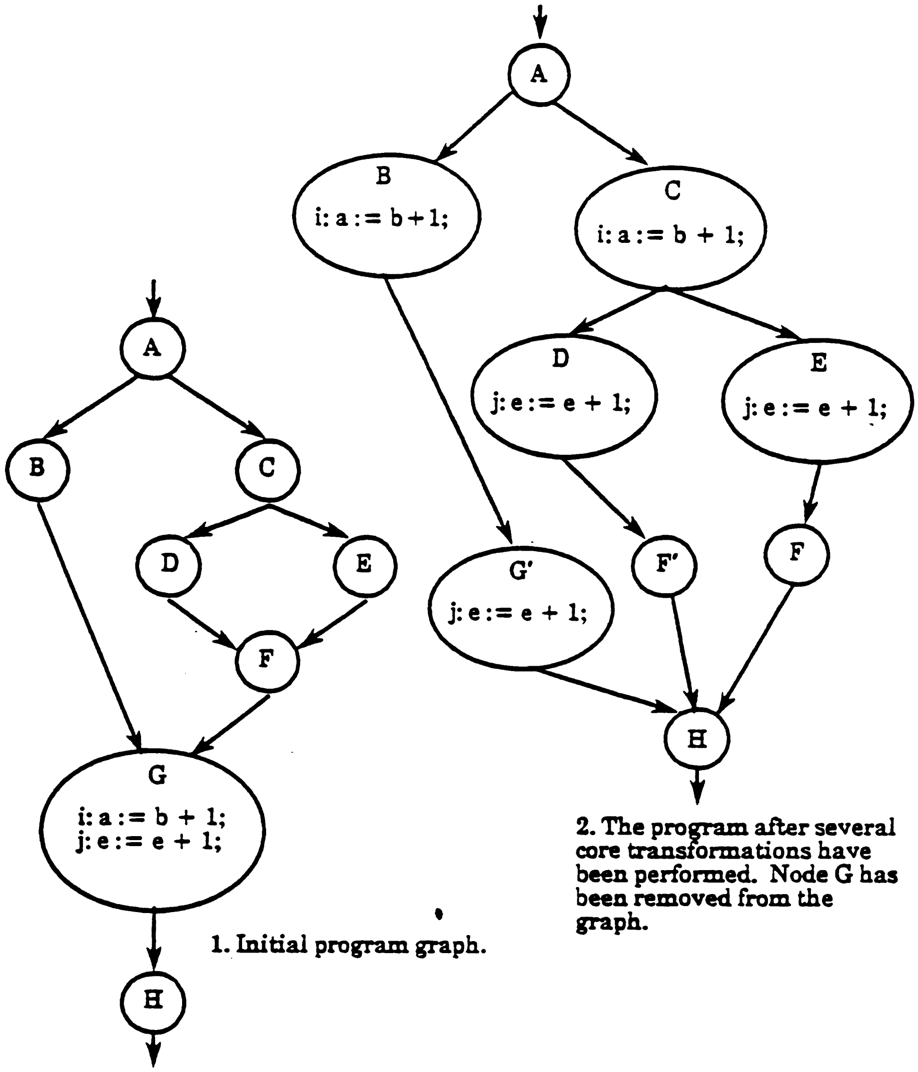


Figure 7: Unification Example

```
I := { i };
repeat
   (* we assume that I always contains all copies of instruction i *)
   1. While ∃j ∈ I s.t. j can move from node(j) to some predecessor X of node(j) and
         node(j) is the only successor of X
         do move-op(j,node(j),X);

   2. If ∃j ∈ I s.t. j can be moved to some predecessor X of node(j) via unification
         then
         select j s.t. the depth of node(j) in the graph is maximized;
         unify(j,node(j),K);
until 2 fails.
```

Figure 8: Migrate Transformation

then the operation is a unification and there is no k in C(i,t) such that node(k) is in reach(X) and k can move to some predecessor of node(k) in reach(X).

Fig. 8 gives a high-level description of *migrate*.

The restriction of *migrate* to acyclic graphs is not acceptable. Fortunately, there is a simple extension which works for reducible graphs as well. Loops pose a problem because operations inside a loop body cannot be removed from the loop by the core transformations alone; whenever an operation is moved outside of the loop the node will be copied on the back edge. To overcome this, we combine *migrate* with standard techniques to remove loop invariant code. The modifications to *migrate* are (with some special cases omitted for clarity):

1. An operation which is initially in a loop L cannot move past the entry node of L.

2. Let i be an operation which is initially outside of a loop L and during the course of the algorithm moves to a node X from which it can enter L. If i is loop invariant with respect to L it is removed from X (splitting and copying X if necessary to preserve other paths) and inserted immediately before the entry point to L. If i is not loop invariant, then it is not permitted to move into L.

The first condition prevents operations in a loop body from being moved indefinitely around the back edge of the loop. We assume, without loss of generality, that all such operations cannot be moved outside of the innermost loop containing them. (In our system, loop invariant code removal is used as a pre-processing step.) Condition two prevents operations from moving into loops from which they cannot subsequently be removed. This avoids lengthening loop bodies unnecessarily.

Because conditionals are never unified, a much simplified version of *migrate* can be written for conditional jumps.

We have also developed transformations that compact the entire program graph., i.e., fully compile it. *Compact-blocks* is one such strategy. The idea is very simple: within each single-entry single-exit block of code perform as much compaction as possible. No unifications are performed and no instructions are copied. It is well-known that the speedup achievable by exploiting parallelism within basic blocks is small [19]. In fact, *compact-blocks* is intended for use primarily as a first step in the compaction process. Its application considerably reduces the number of nodes in the graph without moving any operation to a point where it blocks–due to a data dependency–an operation that could otherwise move.

Compact-path is a global heuristic which moves operations on the "most-important" path in a program graph. We assume that for each conditional j we have two real numbers true(j) and false(j) representing the probabilities that the true and false branches of j will be followed respectively. In many cases such analysis can be performed automatically with good results [19]. We can extend this idea to a DAG of conditionals, where the probability that a certain path will be selected through the DAG is the product of the probabilities of the edges on the path. In the algorithm for *compact-path* (see Fig. 9) p(X,Y) denotes the probability that program execution continues with node Y after the execution of node X.

The user has the option of selecting the path for *compact-path*. For a typical program, the user first explicitly selects the critical paths through the code for optimization. The system then selects and compacts less important paths automatically.

Compact-path can be viewed as a generalization of the technique of trace-scheduling developed in the ELI project at Yale [8]. Trace-scheduling also selects the "important" path (or "trace") through the program and compacts it. However, trace-scheduling does not perform

```
X := root node of program graph;
while ∃ successor of X do
   begin
      for i an operation in X do
         migrate(i);
      X ← max           p(X,S);
          S ∈ successors(X)
   end;
```

Figure 9: Compact-path Transformation

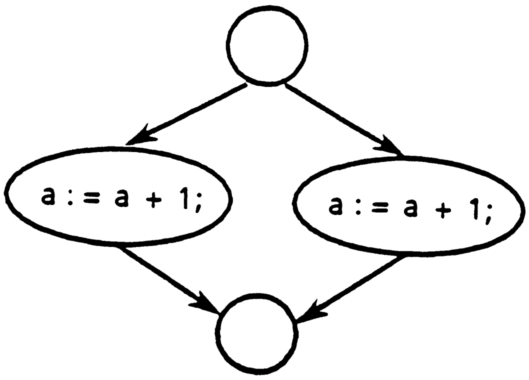


Figure 10: The operation cannot move unless a unification is performed.

unifications and does not allow traces to be merged or altered in any way after compaction. Fig. 10 provides an example for which compacting a single trace without unification results in no improvement. A further advantage of *compact-path* is the tendency of unifications to minimize code explosion. Trace-scheduling introduces code at the entry and exit points of the trace to preserve semantics. When subsequent traces are selected for compaction, this fix-up code cannot move back onto the original trace. With *compact-path* there is some chance that copied instructions can subsequently be unified, thus limiting the size of the final code. This can be a major advantage, as for example in microcode compaction, when the size of the available microstore is small.

6.3 Enabling Layer

These transformations allow the core transformations to be used on general program graphs, and enhance the ability of the core transformations to exploit coarser parallelism.

Cyclic Movement: The core transformations do not (and should not) know about cyclic graphs, as they only deal with adjacent nodes. This transformation allows them to be applied to cyclic graphs in a controlled fashion, by introducing two dummy nodes, Start and Stop, in each cycle. These are dummy nodes in that they do not appear in the final code, and their sole function is to control the application of the primitive transformations.

The point where the dummy nodes are introduced is irrelevant from the perspective of the core transformations.[8] Thus we can choose the point where the dummy nodes are introduced to minimize the length of dependency chains on the path from the Start to the Stop node. This can drastically change the appearance of a loop. The idea is to enable the best parallelization on the code (between the dummy nodes) that forms the reordered loop body (see example in section 8).

Loop Quantization: This technique enables PS to use the core transformations to achieve good results even when other transformations (e.g., loop interchange) may not apply (see section 8).

If full unwinding of all nested loops were possible then we could achieve optimal speedups, subject to data-dependencies, and given enough processors, since no artificial constraints would be introduced by the iteration order of the loops. This is not usually possible due to resource limitations[9]. Still, we would like to get the same effect, albeit on a smaller scale, as if all loops were unwound. Unwinding only one loop is not satisfactory, since it exposes parallelism only in that loop. The idea is to (progressively) unwind a few iterations of all nested loops. The unwinding is constrained by three factors: parallelism available (we want to maximize the amount of parallelism exposed—i.e. minimize dependency chains), space considerations and, most important, preserving correctness. The general techniques used are

[8]Placing them on the original loop back-edges would be equivalent to trace scheduling.

[9]Mainly processor limitations—if we were to have enough processing power to execute the unwound loops in parallel, we could probably afford the space.

found in [20]. An example in which Loop Quantization succeeds even though other techniques fail is given in section 8.

Folding and Balancing: To best exploit the parallelism exposed by Quantization, we apply another transformation which is similar to tree-height reduction [14]. This transformation expresses each written variable in terms of initial values, by using repeated back-substitution on the Quantized loop body. It then "balances" the highest tree by using commutativity, distribution, associativity; algorithms for doing this are found in [17] and [4]. Subtrees corresponding to other written variables are identified and reused for expressions requiring the same computations (see example in section 8).

6.4 Meta Layer

In this layer we deal with coarse grained parallelism, (e.g. partially independent loops and modules) Once the lower levels have terminated, nodes are transformed into special nodes, called *meta-nodes*. The model given in section 3 is extended to include these meta-nodes, which are sets of ordinary nodes. The execution semantics of a meta-node is that its components are allowed to execute in any order, with the usual restriction of reads before writes of the same memory location (see section 3). The Delete and Move-op transformations can be applied to meta-nodes, with dependencies being determined by the aggregate dependency of the components. Note that truly independent conditional jumps, resulting from the overlapping loops, can occur at this level. To support such jumps in parallel requires a multi-processor system with independent conditional branching units. At this coarse level of granularity, however, the expense of such a system is reasonable.

Loop/module overlapping: Even when loops cannot be completely merged they may still be overlapped. More specifically, to overlap two loops L_1 and L_2 while preserving correctness the loops must fall into one of the following categories:

1. L_1 and L_2 do not reference the same variables, or, if they do, the references are to totally separate parts of the variables.

2. L_1 and L_2 reference the same variables, but we can overlap the loops so that all conflicting references occur in a safe order; that is, all writes to a variable location S_x in L_1 will occur

before any of the reads to that location in L_2 and all reads to S_x in L_1 will occur before any writes of S_x in L_2. This is achieved by imposing additional constraints on the results of disambiguation. An example of this technique will be given in the next section.

Independent Conditional Handling: Consider the case where truly independent conditional jumps share a node, as for example, when conditional jumps from different loops occur in the same meta-node as a result of the loop overlapping transformation described above. To preserve the execution semantics of the overlapped loops, runtime synchronization can be used to enforce data-dependencies across the loops. Runtime synchronization, however, can be expensive in terms of running time and hardware requirements. Alternatively, we can use no-ops to provide synchronization (assuming that the processors themselves are synchronous). While this may waste some cycles (on the processor executing the no-ops), this will often be preferable to runtime synchronization particularly when the number of no-ops introduced is relatively small compared to the length of the loop. In fact, this could still be used in conjunction with runtime synchronization, to alleviate synchronization traffic at runtime. The technique would be particularly useful for relatively tight loops with essentially independent iterations containing unpredictable jumps with unequal branches—e.g., a parallel sorting algorithm.

6.5 Mapping Layer

The transformations described above expose parallelism and provide a partial order on the issue of operations. The transformed graph can be viewed as the code for an idealized machine in which no resource conflicts ever occur. Obviously this ideal is unrealizable and can only serve as a bound on the effectiveness of the transformations. Executing the resulting code on realistic architectures requires conversion of the ideal schedule to a schedule that recognizes resource limitations. Unfortunately, even for simple architectural models computing an optimal schedule is NP-hard. We are currently implementing the *Mapping Layer* to cope with this problem.

At the heart of the mapping layer we are implementing is a tabular description of the machine for which code is to be generated. It can be modified by the user to match a particular architecture. The description is currently rather simple-minded. In the next stage of our work we plan to use more accurate machine descriptions, to improve the retargetability of the system and extend its ability to integrate and exploit realistic machine characteristics.

The mapping layer deals with resource constraints and timing considerations (e.g., for pipelined, multi-cycle operations). The term resources includes everything that is required to execute an operation: bus lines, registers, functional units, instruction-encoding conflicts, etc. Including resource constraints and variable length operations greatly complicates the conditions under which a transformation is correct. Fortunately, these checks can be done automatically and efficiently. Resource conflicts are treated as additional data-dependencies. An operation is not allowed to move into a node if the node would then require more resources than the machine has available.

Dealing with variable length operations requires modifications in the definitions of the core transformations. Besides the constraints already stated, now an operation may move only when there are no conflicts with operations it overlaps in time.

Representing variable length operations poses a number of design problems. If each node in the graph represents one clock cycle and the stages of the instructions are stored explicitly in the graph, then the size of the graph increases substantially. This is especially true before compaction, when the sequential code must be padded with many empty nodes to ensure that operations do not overlap. Furthermore, if at some point an operation extends past a conditional-jump the stages of the operation must be duplicated on both of the conditional's exit paths. Similarly, moving an operation above a conditional requires duplicating the stages on both paths. We choose to store operations in the graph as before and only expand operations into stages when it is necessary to check the correctness of a transformation. The only other modification is that edges are assigned lengths to represent the number of cycles between instructions.

We now outline an algorithm to check the correctness of moving an operation i into a node M when operations are pipelined.

1. Define the distance of a path between two nodes to be the sum of the edge lengths on the paths. Let length(i) denote the time (in cycles) required to execute operation i. Let max be the length of the longest instruction.

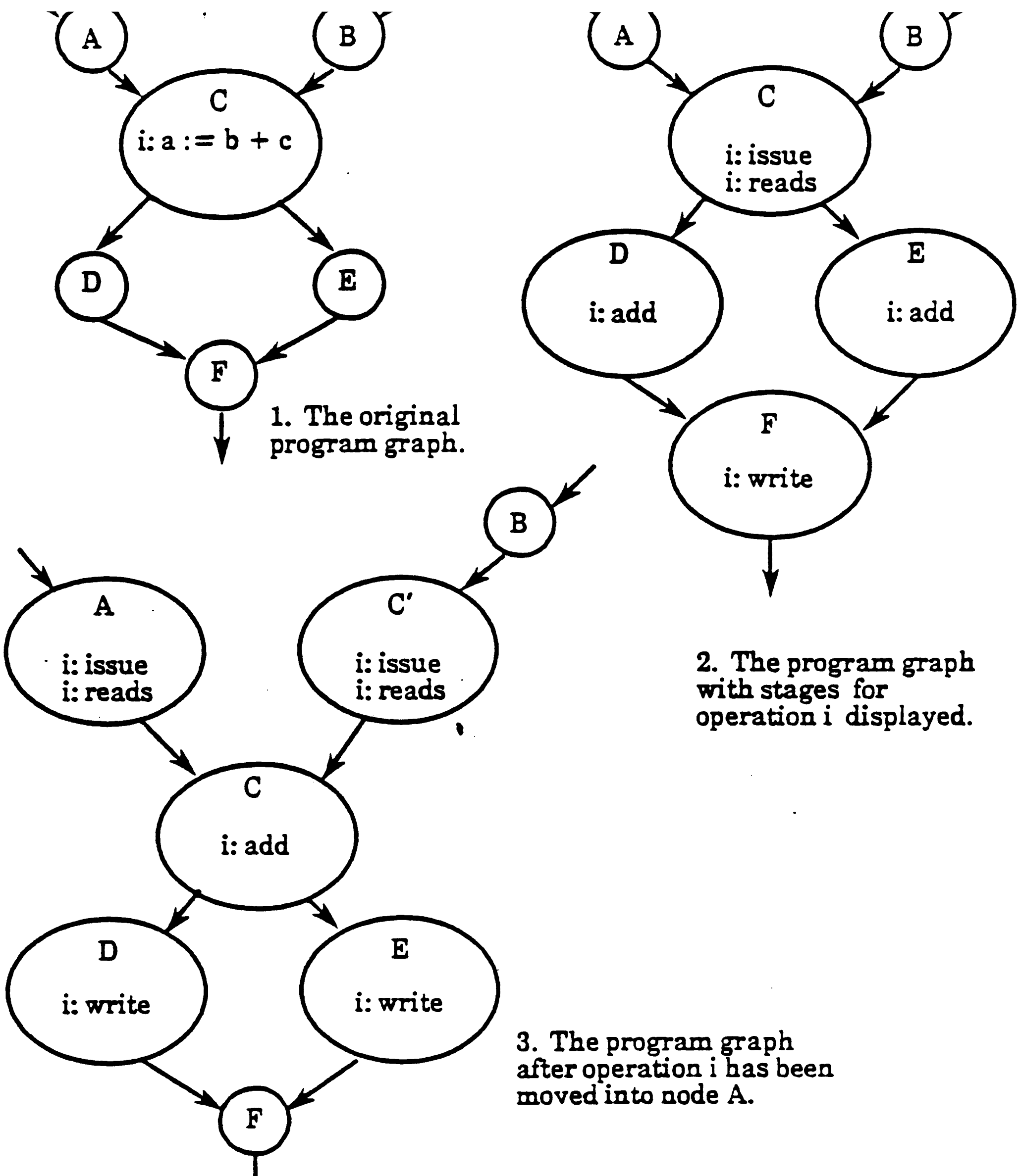


Figure 11: A pipelining example.

2. Beginning at node M, follow all paths leading from M to a distance of length(i). For each operation j in a node on one of these paths expand j into its stages.

3. For each point p on a path and j active at p check the stage of i active at p for conflicts with the corresponding stage of j. If a conflict is detected at any point the transformation fails.

This procedure must then be repeated on all paths leading to M beginning at a distance *max* before M. An example of a *move-op* with stages of the operations displayed is given in Fig. 11.

7 Runtime Synchronization

Despite our best efforts to avoid it by static compile-time scheduling, runtime synchronization will still be necessary if we are to exploit parallelism at the higher levels of granularity (i.e. procedure/module/program). However, at this level of granularity the data-flow approach may prove effective, since the overhead introduced by synchronization is likely to be comparatively small.

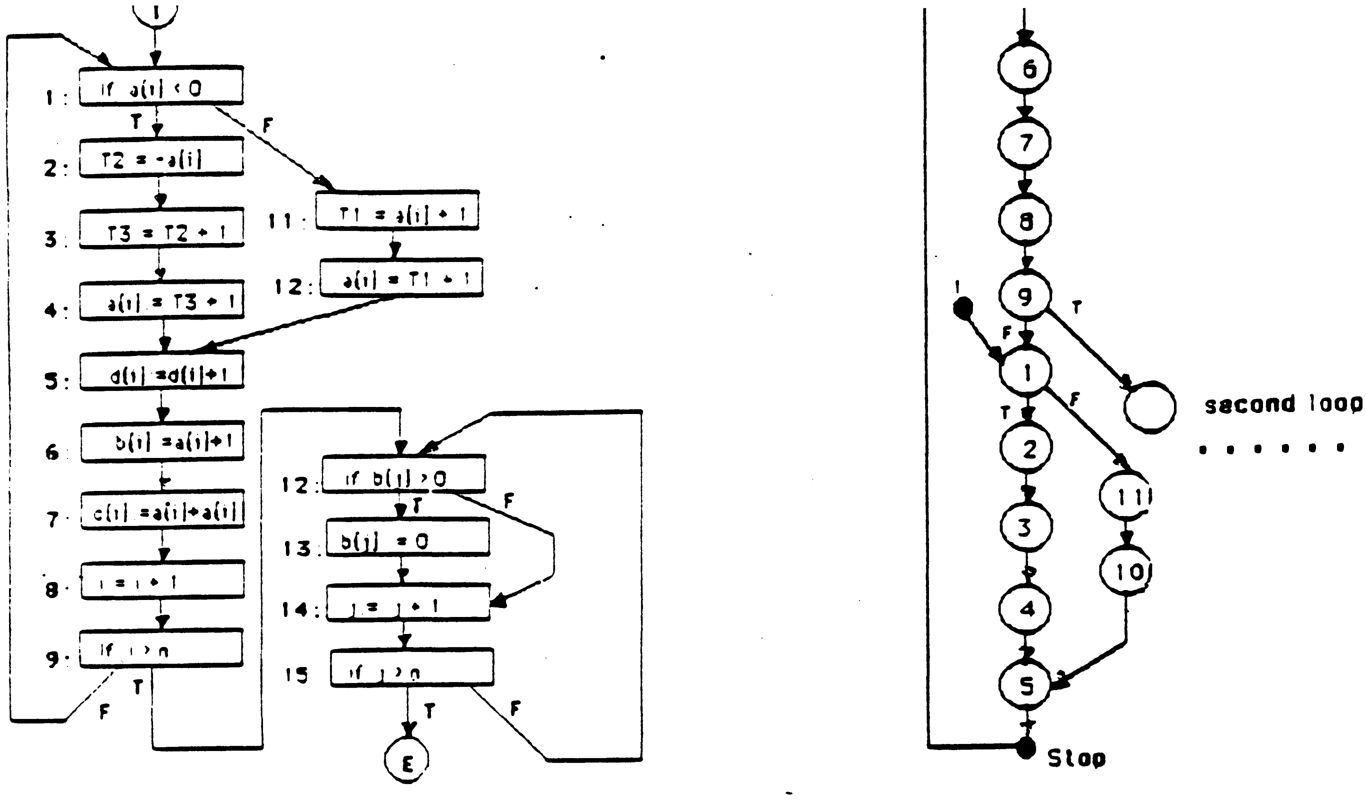
It is too early for us to approach this level in any meaningful fashion. Only after first implementing our prototype and studying its performance will we be able to approach the architectural issues at this level in a non-speculative manner.

8 Sample Uses of Percolation Scheduling

This section gives some examples of the application of Percolation Scheduling as a whole. A detailed description of the process would require a more thorough discussion of the techniques than is possible in the context of this paper. For these examples we assume unit time execution for all operations and no resources conflicts. This is a common idealization, which enables us to obtain a bound on the parallelism our methods can extract. We have also examined the use of PS for real architectures [13] and to more realistic programs (e.g. sorting, some of the Livermore Loops). The preliminary results (obtained by hand-simulation) are very encouraging (speedups of 3 times or more over other compaction techniques). We expect that these speedups will increase as we complete our implementation of PS and are able to handle real (large) programs. We will then be able to present comprehensive and meaningful benchmarks and comparisons.

8.1 Example 1

Consider the sample program in figure 12(a). Figure 12(b) shows the result of the Cyclic Motion transformation (Enabling Level) on the first loop. This transformation is applied before the Core Level processing begins. The first loop is broken after operation 5. This minimizes dependency-chain lengths along the paths created by the introduction of the Start/Stop



(a) (b)

Figure 12: Sample PS Use (a) Original Program. (b) Enabling Transformations.

nodes[10].

Next the core transformations are applied, subject to the Guidance Rules Level and the General Support Level. Operations 1,5,6,7,8,9 have the highest probability of execution and are allowed to move first (Node Probability rule). Notice that several individual operation movements could be done in parallel. Also notice that the probability of the conditional jump branches does not matter as far as the above motions are concerned. Finally, operations from the branches are allowed to move up (Node Probability/Unpredictable Conditionals rules). This completes the optimization of the first loop.

The second loop is similar. The resulting code is shown in figure 13. The code produced thus far is already superior to that achieved by Trace Scheduling. Other methods would also be limited by the control flow and the granularity of the parallelism involved.

The results of the Meta Level are shown in figure 14. Each node is now made into a meta-node and the core transformations are then applied (for Loop Overlapping). Meta-

[10] Note that the General Support Level is used by all the others, at least indirectly. In particular, disambiguation is helpful here.

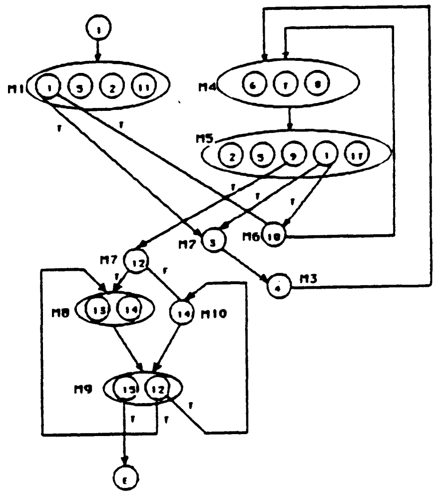


Figure 13: Individual Loops—processed code.

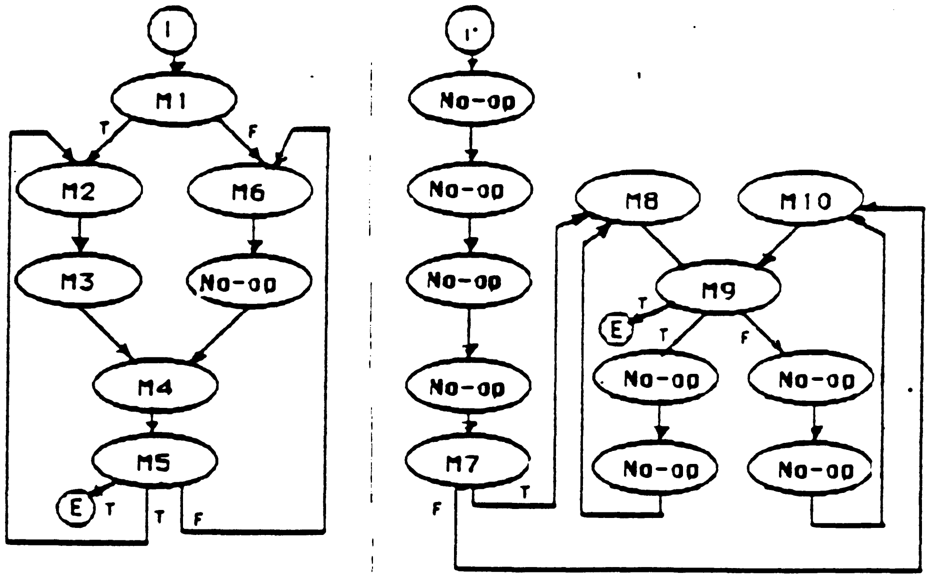


Figure 14: Meta Level—Final schedule.

nodes from the same loop cannot move past each other because otherwise the previous layers would have merged the independent nodes. Thus only meta-nodes from different loops (or non-loop code) can move past each other. In particular, M7 moves up to M5 (i.e., they are merged into a new meta-node). There its movement stops, since M7 has a dependency conflict with M4. The shorter loop is wrapped around the larger one as a result of the Meta transformations. No-ops are used to maintain synchronization between the two overlapping loops throughout the execution (Independent Conditional Handling). Note that while the machines on which the code is run are synchronous, the internal flow of control of the loops is completely separate from each other (i.e., conditionals are completely independent).

8.2 Example 2

As a second example, consider the way Percolation Scheduling techniques deal with recurrences. In figure 15 we find a relatively simple piece of code. Nevertheless, transformations such as loop interchange [3] or vectorization do not apply here. If the loops are in their original order, $X[17,2]$ will be read in iteration $(i = 1, j = 2)$, and written in iteration $(i = 16, j = 1)$. Since the loop on j is innermost, $(1,2)$ occurs before $(16,1)$ and thus we have a read before a write. Reversing the loops will write into $X[17,2]$ in iteration $(j = 1, i = 16)$ and read from it in iteration $(j = 2, i = 1)$; but since the loops are now reversed, the write will occur before the read, which is obviously not semantically equivalent to the original code. The interchange is therefore illegal. Furthermore, vectorization does not help, even using expansion and loop distribution (breaking): the first statement could be vectorized, but that would only reduce the execution from $4 * n^2$ to $3 * n^2 + 1$ steps, assuming vector processors of size of n, which would be wasteful if n is large. Furthermore this is no better than a traditional multiprocessor, which can also achieve $3 * n^2$ execution steps by running the first two statements in parallel. Even if more than two processors were available on a traditional multiprocessor, attempting to distribute the loop across the processors will incur very heavy communication costs due to the tight dependencies between the iterations.

Percolation scheduling, on the other hand will be able to quantize the loop by unwinding on both i and j in such a manner as to efficiently utilize the resources. Note that we can do this quantizing even though loop interchange is impossible. This is because quantization,

```
   Do i=1,n
   Do j=1,n
(1)  T1=X[i+16,j];
(2)  T2=X[i+1,j];   /* Notice that statements (2 and 4) form a recurrence */
(3)  T1=T1+T2;
(4)  X[i+1,j+1]=T1;
   Od; Od;
```

Figure 15: Original Recurrence code

unlike interchange, preserves the relative order of execution of the statements inside the "box". By comparing

$$(1)\ T1 := X[i+16,j]; \ldots\ldots\ldots\ldots (4)\ X[i'+1,j'+1] := T1;$$

the disambiguator can determine that conflicts between iteration (i,j) and (i',j') can occur when $i' = i - 15$ and $j' = j + 1$. Using the techniques described in [20], we find that choosing to unwind 15 times ($K_i = 15$) is safe, i.e., will preserve the semantics of the original loops. For this example we will choose $K_j = 15$ as well. The other dependency that needs to be examined to quantize this loop is

$$(2)\ T2 := X[i+1,j]; \ldots\ldots\ldots\ldots\ldots (4)\ X[i'+1,j'+1] := T1.$$

This turns out to introduce no other constraints on the unwinding. The resulting loop is shown in figure 16.

The disambiguation mechanism will decide that the 15 unwindings along i (i to $i+14$) are independent of each other, and can be done in parallel. Applying folding and balancing (tree height reduction) techniques to each such group (for j to $j+14$), we can, resources permitting, execute each unwound iteration in 6 steps: 1 for loading, 4 for computing sums and storing, and 1 for storing the last results (see figure 17). To fully exploit this parallelism, we need 16 processors per unwinding on i, for a total of $15 * 16$ processors[11]. The speedup

[11]By reducing the speedup slightly, from 6 to 8 time steps per unwinding of i, we could do with only $15 * 8$ processors by adding 2 extra steps for initial and intermediate storage of results.

```
Do i=1,n,15
Do j=1,n,15
  T1=X[i+16,j];        /* j+0, i+0  */
  T2=X[i+1,j];
  T1=T1+T2;
  X[i+1,j+1]=T1;
  .................
  T29=X[i+2,j+15];   /* j+15,i+0  */
  T30=X[i+1,j+15];
  T29=T29+T30;
  X[i+1,j+16]=T29;

........................

  T1=X[i+31,j];        /* j+0, i+15 */
  T2=X[i+1,j];
  T1=T1+T2;
  X[i+16,j+1]=T1;
  ..................
  T29=X[i+31,j+15];   /* j+15,i+15 */
  T30=X[i+16,j+15];
  T29=T29+T30;
  X[i+16,j+16]=T29;
Od; Od;
```

Figure 16: Same Recurrence with quantized unwinding

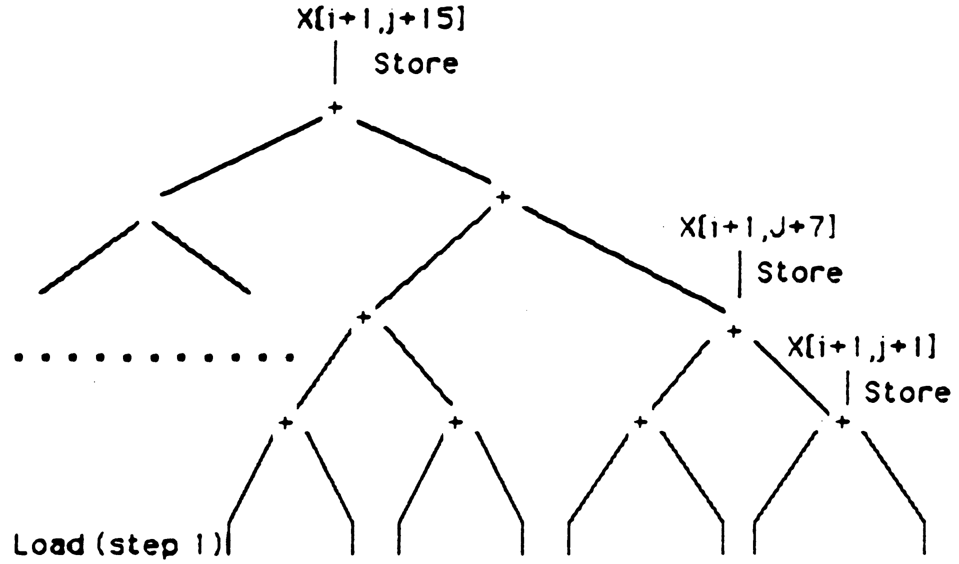


Figure 17: Balanced tree for i, j..j+14

achieved is from $4 * n^2$ steps to $6 * (n/k_1) * (n/k_2)$, (in our example, $k_1 = k_2 = 15$). If more resources were available, we could unwind more, getting even more dramatic speedups. In particular, if the loops were fully unwound the speedups would be optimal. Even when this is impossible, we will still get good performance given the available resources.

9 Implementation

The system actually resides on two machines, a Vax 11/780 and a Xerox Dandelion workstation, and is part of an environment that supports user interaction with the compiler though a graphical interface. The code is written in Franz Lisp (on the Vax) and Interlisp (on the Dandelion). Lisp was selected for its robust environment of system functions and debugging tools. All PS transformations are executed on the Vax. The Xerox workstation serves to display the program graph and profiler information and to accept user input.

The system was placed on two machines to take advantage of the graphics capabilities of the Dandelion. Nearly all user commands are issued with the Dandelion's mouse. A typical sequence is to click on a portion of the code, and select "compact" from a menu of options. The system then compiles the required code fragment (or the whole program, if requested), and updates the profiling information to reflect the improvement. When ambiguities which cannot be resolved automatically prevent some potentially beneficial transformation, the system may ask for specific additional information or user help in resolving the ambiguity. This allows for incremental compilation under user control. Alternatively, the system can compile the whole program automatically, without any user interaction.

The high-level source language currently used by the environment is Tinylisp, a Fortran/C-like language with lisp syntax. The machine-level language is called naddr. Naddr supports typical machine operations. The choice of language is of no concern to the system proper; the compiler is only concerned with dependencies and flow of control. The model of computation described previously is flexible enough to allow complex constructs (i.e., arbitrary multi-way jumps, any number of write/read variables in an instruction, etc.). Tinylisp and naddr were chosen to ease implementation and because of our previous experience with them. in the ELI project [6]. This has allowed us to concentrate on the problems of interest - the parallelization transformations. As our work progresses we plan to convert to other, more standard languages.

We are currently porting the system to Common Lisp on a Symbolics lisp machine. The present system is limited both in speed and in the size of what can be compiled. We expect the use of the lisp machine will alleviate these problems completely.

Common Lisp was chosen to make the new system as easily portable as possible. We believe the PS compiler should serve both as a code development tool and as a research vehicle for investigating new code transformations. Therefore, experimentation with the system beyond the designers' group is desirable and portability for wide distribution is a main concern.

10 Early Experimental Results

Our system has recently become operational and has been used to generate code for several test programs. Some of the results are presented in Table 1. The figures in the table were obtained with completely automatic compaction. The programs in our test suite were chosen for several reasons. While small, they perform important functions that are likely candidates for microcoding, at least for machines geared towards numerical programming. Of course, due to their importance much effort has been expended on developing good algorithms and hand-coding the resulting microprograms for efficient execution. The results are not meant to impress by absolute performance, particularly since we intentionally chose to use as a starting point standard algorithms which do not yield much parallelism. The examples presented here are meant to demonstrate the viability of our approach even for programs exhibiting low inherent parallelism (*ln*, *sqrt*, *prime*, *fft*) and unpredictable flow of control (*trcl*, *sort*, *ll24*). Results for *matmul* have been included to show what can be achieved when the code has a large amount of exploitable parallelism. Even in the automatic mode our system obtains speedups at least as good and sometimes better than other approaches.

We define the speedup (%) achieved by our environment as the ratio:

$$Speedup = \left(\frac{executed\ useful\ sequential\ operations}{executed\ parallel\ instructions} - 1\right) * 100.$$

Useful operations are defined as all arithmetic, logical, flow-of-control, and indexed load/store operations. They do not include register-to-register moves and direct loads and stores. This is very generous to the sequential version of the code and is essentially equivalent to assuming ideal global register allocation and perfect pipelining of moves and direct stores/loads. In the parallel version all executed microinstructions are counted. Thus a load/store or move will be done "for free" only if it is actually overlapped with some other useful operation in the compacted code. The numbers in Table 1 reflect this conservative model. If register-to-register moves and direct loads/stores are counted as executed operations in the sequential model, the speedups obtained by our system are two to four times larger on average than those presented in the table.

We believe that the dynamic measurement of speedup described above is more objective than the traditional static measure which compares the size of the original and compacted

Table 1: Some results.

program	*description*	*unwinding*	*speedup %*
dotprod	inner product	1	100
dotprod	inner product	2	200
dotprod	inner product	4	398
fft	fast fourier transform	1	204
fft	fast fourier transform	2	258
ll24	vector minimum	1	60
ll24	vector minimum	3	292
ln	natural log	1	98
ln	natural log	4	208
matmul	matrix multiply	3	268
matmul	quantized matrix multiply	3 x 3	1004
prime	prime sieve	1	62
prime	prime sieve	2	137
sort	insertion sort	2	170
sqrt	square root	1	86
sqrt	square root	2	86
trcl	transitive closure	1	92
trcl	transitive closure	3	423

code. Typically, whenever the code contains any conditional-jumps other than the exit-tests for loops a static measure yields better speedups (by as much as a factor of two) than we report here.

The small size of the code, combined with the above constraints, make the results in Table 1 an empirical lower-bound on the performance we can expect from our environment. In this light we consider these results to be extremely encouraging. Indeed, even for apparently sequential code and despite the limitations placed on the system by our current hardware/lisp configuration the increase in speed can be quite dramatic. For example, *trcl*'s original (already good) speedup of 92% more than quadrupled with a small unwinding of the innermost loop. On average we achieved 450% speedup for the largest unwindings with completely automatic compaction. Further interactive compaction can improve the automatic results by about 300

The experiments we discuss here were primarily designed to test the effectiveness of our transformations and their range of applicability. As such we did not place restrictions on the number of resources used in any horizontal instruction. Nevertheless, we are encouraged by the relatively small amount of resources used (typically under ten operations per cycle). Occasionally large numbers of operations per instruction are generated at the beginning of the compacted code, but these can usually be spread out without a significant decrease in the speedup obtained. The whole process of resource management–either automatic or user controlled through transformations described previously–may actually increase the speedups reported here by allowing further overlap of stages of operations from different instructions. Further experimentation is necessary before any conclusions can be reached on the positive or negative impact of such transformations on speedups.

Inspection of the code reveals that further speedups will be possible in almost every program as soon as the conversion to Common Lisp is complete and we can take advantage of the larger space and speed available to perform larger unwindings, coupled with the loop-quantization and cycle-breaking techniques described previously. While these techniques are implemented in our system, the limitations of our current configuration do not allow us to significantly exploit their potential.

11 Acknowledgements

Alex Aiken's contributions to both the implementation and design of the system have been invaluable for the progress of the PS project. The technical and editorial comments of my colleagues Alan Demers, David Gries, Kevin Karplus, Hal Perkins, Fred Schnider and Jon Solworth are gratefully acknowledged. Their help has significantly improved this paper.

References

[1] A.V.Aho and J.D.Ullman Principles of Compiler Design. Addison-Wesley, 1977.

[2] J.R.Allen and K.Kennedy PFC: a program to convert Fortran to Parallel form. Rice University Technical Report MASC TR 82-6, 1982.

[3] J.R.Allen and K.Kennedy Automatic Loop Interchange. In the Proceedings of the Symposium on Compiler Construction, SIGPLAN Notices Vol.19, No.6, 1984.

[4] R.Brent The Parallel Evaluation of General Arithmetic Expressions. JACM 21, pp. 201-206, 1974.

Proc. IFIP Congress '80, pp. 175-180, 1980.

[5] J.J.Dongarra, F.G.Gustavson and A.Karp Implementing linear algebra algorithms for dense matrices on a Vector Pipeline Machine. SIAM Review, Vol. 26, No. 1, 1984.

[6] J.A.Fisher, J.R.Ellis, J.C.Ruttenberg and A.Nicolau Parallel Processing: A Smart Compiler and a Dumb Machine. Proc. of the ACM Symposium on Compiler Construction, 1984.

[7] J.A.Fisher An Effective Packing Method for Use with 2^n-way Jump Instruction Hardware. Proc of the 13th Annual Microprogramming Workshop, SIGMICRO, 1980.

[8] J.A.Fisher Trace scheduling: A technique for global microcode compaction. IEEE Transactions on Computers, Number 7, pp. 478-490 1981.

[9] W.H.Harrison. *Compiler Analysis of the Value Ranges for Variables.* IEEE Transactions on Software Engineering SE-3(3), pp.243-250, 1977.

[10] K.Kennedy Automatic translation of Fortran programs to vector form. Rice University Technical Report 476-029-4, 1980.

[11] D.D.Gajski, D.A.Padua, D.J.Kuck and R.H.Kuhn A Second Opinion on Data-Flow Machines and Languages. IEEE Computer, Vol. 15, No. 2, 1982.

[12] D.Jacobs, J.Prins, P.Siegel, K.G.Wilson Monte Carlo Techniques in Code Optimization. Proceedings of The Micro-15 Workshop on Microprogramming, ACM/IEEE, 1982.

[13] A.Nicolau and K.Karplus. ROPE: A Statically Scheduled Supercomputer Architecture. Proceedings of the First International Conference on Supercomputer Systems, St.Pettersburg, Fla. December 1985.

[14] D.J.Kuck Parallel Processing of Ordinary Programs. In Advances in Computers, Vol. 15, pp. 119-179, 1976.

[15] D.Kuck, R.Khun, D.Padua, B.Leasure and M.Wolfe Dependence Graphs and Compiler Optimizations. Proceedings of POPL 81, ACM, pp. 207-218 1981.

[16] H.T.Kung. Let's design Algorithms for VLSI Systems. Proceedings of the Conference on Very Large Scale Integration: Architecture, Design, Fabrication. California Institute of Technology, pp. 65-90, January 1979.

[17] Y.Muraoka, Parallelism Exposure and Exploitation in Programs. University of Illinois, Urbana, Dept. of Computer Science, Tech. Rep. 71-424, 1971.

[18] Nicolau, A. Parallelism, Memory Anti-Aliasing and Correctness for Trace Scheduling Compilers. Yale University Ph.D. Thesis, 1984.

[19] A.Nicolau and J.Fisher Measuring the Parallelism Available for Very Long Instruction Word Architectures. IEEE Transactions on Computers (November 1984 Issue), 1984.

[20] A.Nicolau Loop Quantization, or Unwinding Done Right. Cornell University, Dept. of Computer Science Technical Report, TR85-709, October 1985.