A Development Environment for Horizontal Microcode

Alexander Aiken*
Alexandru Nicolau†

TR 86-785

September 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

*Supported in part by the Cornell NSF Supercomputing Center and an IBM fellowship.
†Supported in part by NSF grant DCR-8502884 and the Cornell NSF Supercomputing Center.

A Development Environment for Horizontal Microcode

Alexander Aiken*   Alexandru Nicolau†

Department of Computer Science
Cornell University
Ithaca, NY 14853

Abstract

This paper describes a development environment for horizontal microcode. The environment uses Percolation Scheduling–a transformational system for parallelism extraction–and an interactive profiling system that gives the user control over the microcode compaction process while reducing the burdensome details of architecture, correctness-preservation, and synchronization. Through a graphical interface the user suggests what can be executed in parallel, while the system performs the actual changes using semantics-preserving transformations. If a request cannot be satisfied, the system reports the problem causing the failure. The user may then help eliminate the problem by supplying guidance or information not explicit in the code.

Index Terms–microcode, compaction, Percolation Scheduling, environment, transformation, parallelization, compiler

*Supported in part by the Cornell NSF Supercomputing Center and an IBM fellowship.
†Supported in part by NSF grant DCR-8502884 and the Cornell NSF Supercomputing Center.

1 Introduction

This paper describes a development environment for microprograms. The environment consists of a hierarchy of parallelizing transformations, an interactive profiler, and a graphical interface. The components of the system, their use, and future work are discussed.

Our ultimate goal is to automatically generate better code than can be produced by human experts. However, due to the complexity of code-generation problems, a compiler must rely on heuristics which sometimes fail to produce optimal or nearly optimal code. Furthermore, the compiler's analysis of a program usually cannot capture the user's knowledge of the general problem; thus the user may be able to make decisions based on information not available to the compiler. The support environment we are building is designed to allow the user to control compaction and provide an integrated interface through which additional information that may assist in the optimization process can be supplied. In our system, the role of the compiler is to exploit the easily extractable parallelism. While this may suffice, the user can "fine-tune" the code for better performance. The other components of our system, the profiler and graphical interface, are being designed to support this activity. The need for such interactive compilation has been widely recognized. An in-depth discussion of the desirability of such a system and its potential advantages is found in [Weg86].

The current trend toward larger and more complex microprograms and the development of techniques such as dynamic microcoding [WC86] increases the need for microcode development tools [DS78]. RISC machines, array processors, and VLIW machines are programmed directly in microcode, and CISC machines have large microcode programs that interpret higher-level machine instructions.

The environment we propose maps programs written in a high-level language onto horizontal microengines. A first version of our environment will generate code for the current IBM/FPS-264 Production Supercomputer, part of the NSF Supercomputing Center at Cornell, as well as more conventional microengines.

Existing high-level language compilers for parallel machines do not provide the needed support for exploiting parallelism in microcode. Important advances in parallelizing ordinary code have been achieved [AK82,Fis77,KKP*81]. Interesting work has also been done in the development of environments for supporting parallel computation [HK84,Sny83]. However, this work has dealt with coarse-grained parallelism and has provided support in configuring pre-optimized modules into coherent concurrent systems. Because the parallelism-extraction of current compilers is too coarse, humans are generally much better at microcode compaction than available systems. Thus, in practice, microcode is still compacted by hand when speed is critical. We are proposing a system that supports semi-automatic extraction of fine- and coarse-grained parallelism in a uniform environment. The mundane aspects of parallelization (i.e., ensuring the preservation of semantics) are fully automated. The system will eventually incorporate knowledge of the specific parallel machine for which code is generated, freeing the user of the need to be intimately familiar with low-level details. Using this environment we hope to achieve code quality comparable or even superior to that achieved by expert hand-coding in much less time.

A typical interactive session proceeds as follows. The user first requests some aggregate (global) transformation of the code. Then, with the help of profiler information, the user refines the code by requesting specific transformations. When such a request is made, the system tries to instantiate it by a series of semantics-preserving transformations. If the instantiation succeeds, the code is changed accordingly. Otherwise,the system reports the cause of the failure, e.g., a dependency would be violated. The environment's diagnosis will usually be accurate. Sometimes, however, the system may not be able to achieve the desired results, particularly when the heuristic application of several transformations is involved. In such cases the user may guide the system through a sequence of lower-level transformations that could achieve the desired result while still guaranteeing correctness. For example, transformations sometimes fail due to the inability of the system to eliminate spurious dependencies. Two indirect references could appear to refer to the same memory location–thus causing a dependency–when in fact the references are distinct. The user may realize this based on information available from the problem domain but not explicit in the code. In this case the user may choose to ignore the conflict and direct the system to perform the transformation. When dependencies are transient (i.e., two indirect references conflict for only some of their possible indexes) or if the user is not certain that the conflict detected by the system is spurious, he may request that the system proceed with the transformation and provide a safe runtime escape route. Finally, the user may change the code arbitrarily, outside the transformations provided by the system. Of course, here the environment cannot guarantee the preservation of semantics. In this context, our work can be seen as complementing that of formal verification of microcode [MD86].

At the heart of our environment is Percolation Scheduling (PS), which developed out of our experience with Trace Scheduling in the ELI project at Yale [FERN84]. PS is a hierarchy of semantics-preserving transformations that convert an original *program graph* (control-flow graph) into a more parallel one. PS globally rearranges code in an attempt to exploit parallelism. Its core consists of a small set of primitive program transformations; the transformations are atomic and thus 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 to exploit coarser parallelism.

Aided by the higher levels of the hierarchy, the core transformations operate uniformly on an entire program graph. The transformations can also be applied to partially compacted 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.

2 Architectures

Several existing architectures could benefit from our environment. Horizontal microengines are the obvious candidates. Even non-parallel architectures such as vertical lookahead (pipelined) engines could use the large numbers of sequential operations clustered together by percolation scheduling to efficiently fill pipelines. Hardware to handle multiple conditional-jumps can also be effectively utilized in our environment. The design of such a hardware mechanism and its advantages are described in [KN85].

Data-flow microengines are also suited to take advantage of our system [PHS85]. Traditionally, it has been claimed that data-flow architectures require very little compile-time analysis. From a pragmatic point of view, however, this lack of compile-time effort imposes a very heavy burden in terms of communication and synchronization costs and leads to extremely inefficient use of memory and resources [GPKK82]. Through PS transformations a correct partial order for the issuing of operations can be obtained at compile time and a reasonable partition of the program and data between the various functional units can be achieved. This could significantly reduce runtime communication and synchronization needs as well as the lengths of queues of waiting operations. The atomic nature of the core transformations and their independence makes PS attractive for data-flow compilers and for execution on data-flow machines.

Statically scheduled multiprocessors such as the FPS-264, FPS-164, Mars 432, and the ELI-512 could also benefit from the use of the proposed environment and techniques.

3 The Core of Percolation Scheduling

The core transformations are easy to understand and implement and are independent of any heuristics. They are the lowest layer in the hierarchy of transformations and guidance rules. Higher levels of this hierarchy direct the core transformations and rearrange the program graph to allow more code motion by the core transformations.

In the following sections we present an overview of the Percolation Scheduling hierarchy and the work we have completed. In these sections, the term *node* (in a program graph) refers to a microinstruction. An operation is a component of some microinstruction. In the examples, lower case letters denote operations and capital letters denote nodes. To simplify illustrations, nodes attached to edges entering or exiting the subgraph of interest are not shown. These nodes are denoted by "I_j" (for incoming edges) or "E_k" (for exiting edges).

Four primitive transformations defined in terms of adjacent nodes in a program graph form the core of PS. Repeatedly applying the transformations allows operations to "percolate" towards the top of the program graph from the various parts of the code–hence the name Percolation Scheduling. Operations are packed together in nodes as PS is applied to a program graph, yielding more efficient microcode.

The details of the transformations deal with maintaining the integrity of all affected paths. A brief description of each transformation is given below. A formal description of the model of computation as well as rigorous definitions of the core transformations and proofs of correctness can be found in [Nic84c].

3.1 Delete Transformation

A node in the program graph can be removed by the *delete* transformation when the node contains no operations or when it becomes unreachable. Nodes without any operations may occur as a result of other transformations or as part of the original program graph. An empty node does not affect the execution semantics of the program in any way and may be deleted, provided the outgoing edges of its predecessors are reset to point to the deleted node's successor. This will preserve the semantics of the original program. An illustration is given in Fig. 1.

An unreachable node is a node other than the start node which has no predecessors. Such a node is clearly unnecessary and may be deleted from the graph.

3.2 Move-op Transformation

The *move-op* transformation moves an operation that does not affect the flow of control from a node N to a node M through the edge (M, N) provided no data-dependency exists between operations in M and the operation being moved. Care must be taken not to affect the computation of paths passing only through N but not through M. To ensure this, the paths are split and provided with a copy of the original N. An illustration is given in Fig. 2.

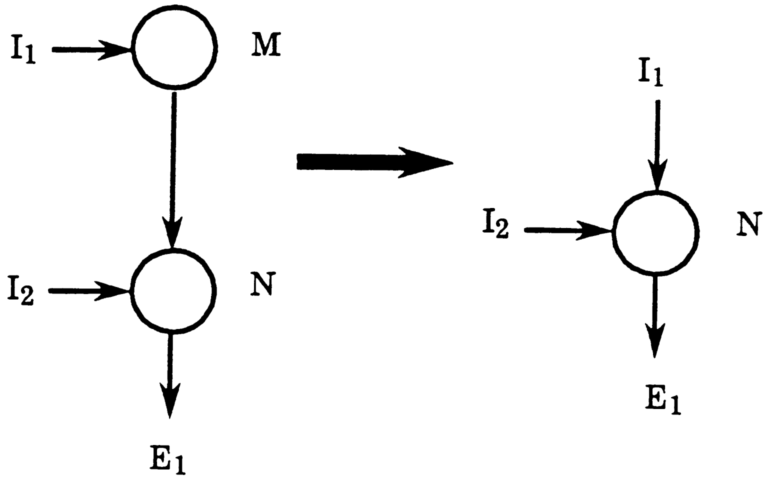


Figure 1: Delete Transformation

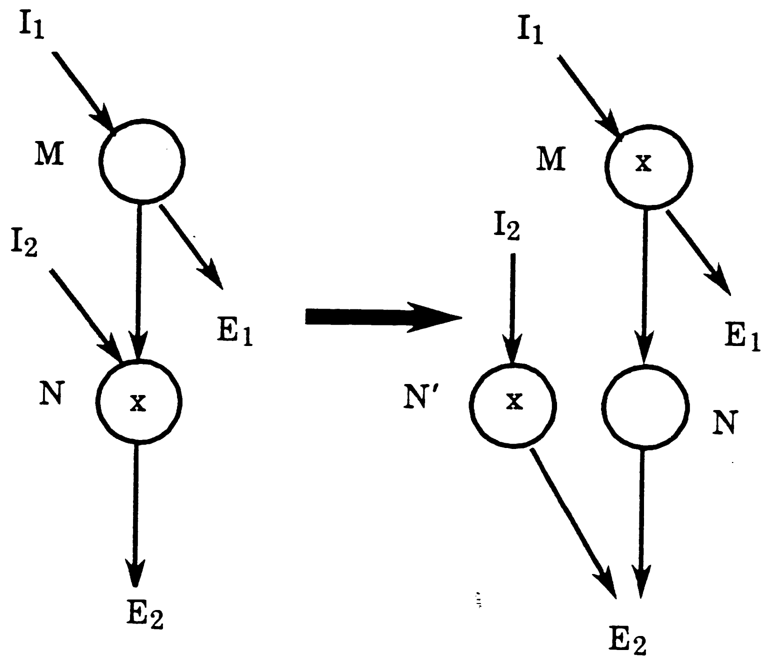


Figure 2: Move-op Transformation

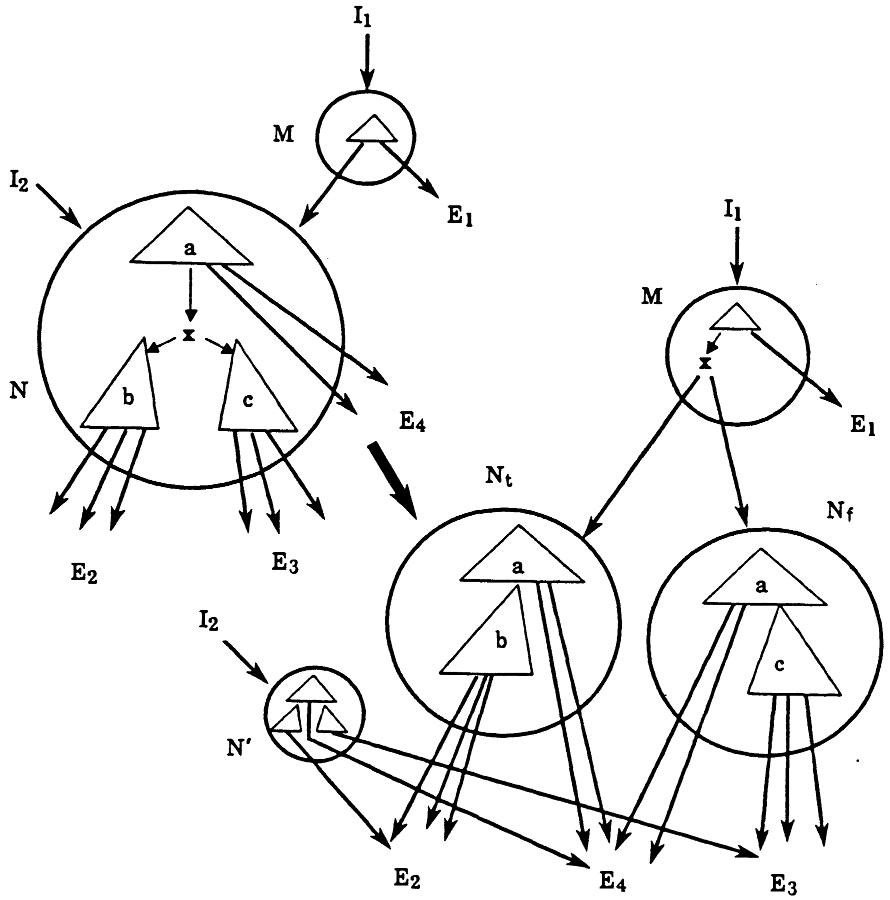


Figure 3: Move-cj Transformation

3.3 Move-cj Transformation

The *move-cj* transformation moves a conditional-jump x from node N to node M through an edge (M, N) provided that no dependency exists between M and the component being moved. Paths passing only through N but not through M must not be affected. To ensure this, the paths are split and N is copied. Because we allow an arbitrary rooted DAG of conditional-jumps in a node and the conditional-jump being moved may come from an arbitrary point in that DAG, N will be split into N_t and N_f, where N_t and N_f correspond to the true and false branches of the moving conditional. An illustration of the transformation is given in Fig. 3. In the illustration, a represents the DAG of conditionals (in N) not reached by x, b represents the DAG of conditionals reached on x's true branch, and c the DAG of conditionals reached on x's false branch. N' is the copy of N.

A detailed description of a hardware mechanism that efficiently implements general conditional-jump DAGs is found in [KN85]. While a multiway jump mechanism will take full advantage of the power of PS, it is not required for the use of our system. The environment can be used to generate good code for any horizontal architecture.

3.4 Unification Transformation

The *unification* transformation moves a unique copy of identical operations from a set of nodes $\{N_0, N_1, N_2, \ldots\}$ to a predecessor node M. This is done only when no dependency exists between M and the component being moved and when the edges (M, N_i) exist for all nodes in the set.

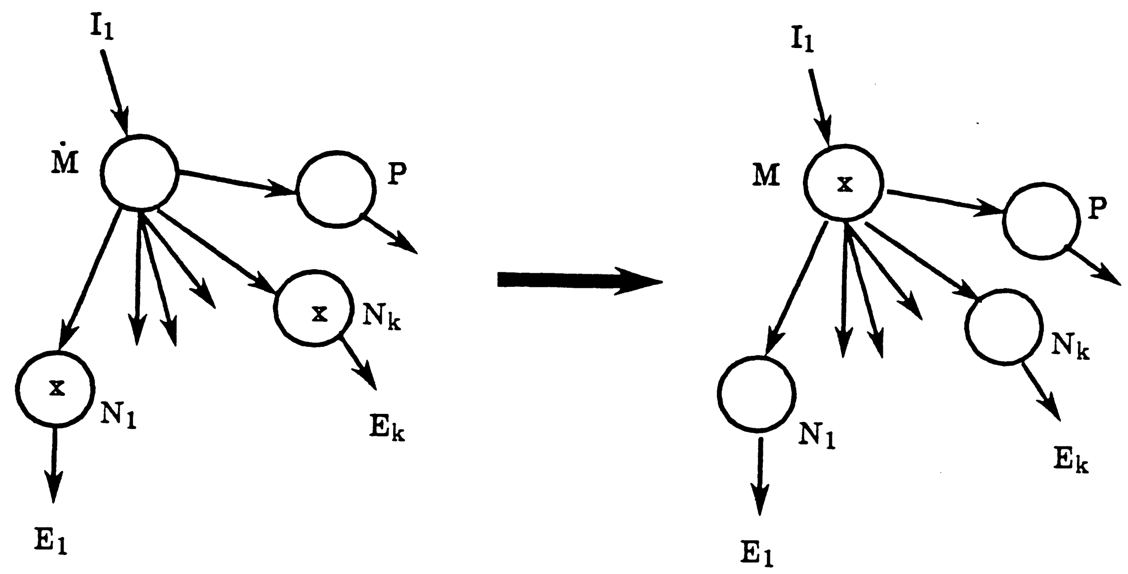


Figure 4: Unification Transformation

Paths passing through N_i but not through M must not be affected–as usual, splitting and copying is used. An illustration is given in Fig. 4.

3.5 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.

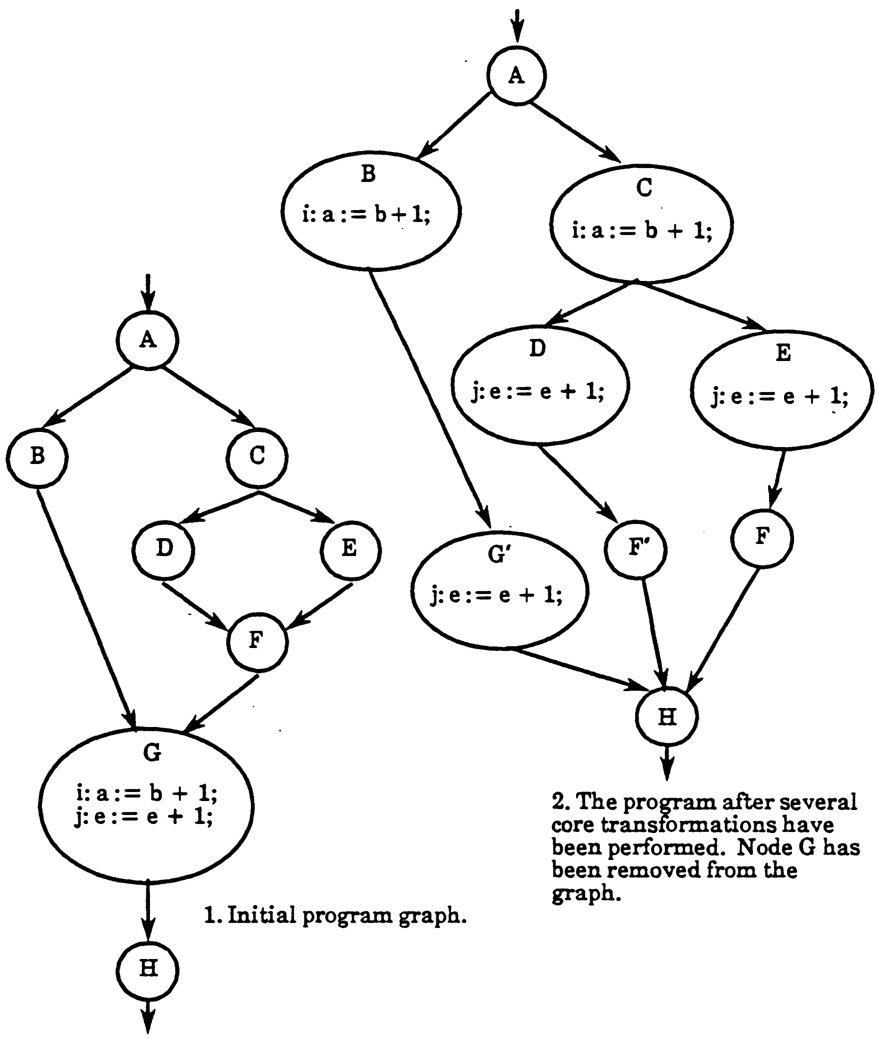
4 Beyond the Core Transformations

The core transformations are very low-level. Even for small examples the number of transformations required to compact the graph of a microprogram is considerable; it would simply be too tedious for a user to issue them one at a time. What is required is a set of higher-level transformations; we partition these transformations into two levels: *scheduling transformations* built on the core transformations that actually compact the program graph and *enabling transformations* which rearrange the program graph to expose parallelism.

4.1 Scheduling Transformations

A simple transformation that we have implemented in the environment, 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



1. Initial program graph.

2. The program after several core transformations have been performed. Node G has been removed from the graph.

Figure 5: Unification Example

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. 5 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).

```
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 6: Migrate Transformation

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, 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. 6 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.

$X :=$ root node of program graph;
while $\exists$ successor of X **do**
  begin
    for i an operation in X **do**
      migrate(i);
    $X \leftarrow \max\limits_{S \in \text{successors}(X)} p(X,S)$;
  end;

Figure 7: Compact-path Transformation

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. The need for such transformations is clear; it is unlikely that a user will wish or even need to manually apply individual transformations to the entire program. Instead, critical sections of code can be optimized or transformed to expose parallelism, after which global heuristics can be applied to obtain a good schedule.

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 [NF84]. 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 [NF84]. 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. 7) $p(X,Y)$ 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 [Fis77]. Trace-scheduling also selects the "important" path (or "trace") through the program and compacts it. However, trace-scheduling does not perform unifications and does not allow traces to be merged or altered in any way after compaction. Fig. 8 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.

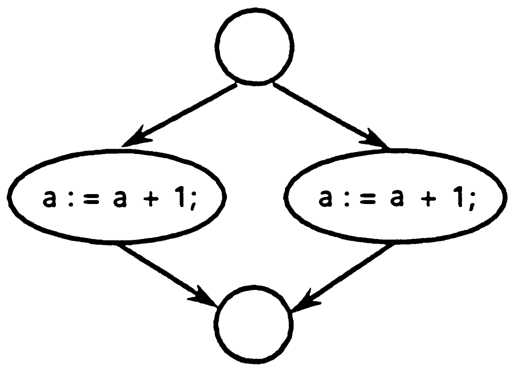


Figure 8: The operation cannot move unless a unification is performed.

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 is a major advantage in microcode compaction when the size of the available microstore is small.

4.2 Enabling Transformations

The *Enabling* level provides transformations of the program graph that rely on global information and therefore cannot be accomplished by the core transformations. The purpose of these transformations is to expose parallelism for exploitation by the core transformations.

A well-known example of an enabling transformation is variable renaming. The judicious choice of new variable names can often remove dependencies between statements.

The *cycle-breaking* transformation guides the application of the core transformations to loops. Intuitively, *cycle-breaking* "breaks" a loop by picking an edge e across which no operation may move. The loop is shifted to make e the back edge. This requires introducing fix-up code immediately before the loop. The point where a cycle is broken is chosen to minimize the lengths of dependency chains in the resulting loop. The loop body can then be compacted as a straight-line piece of code by the core transformations.

The primary loop optimization tool used in the environment is *loop quantization*, a technique for unrolling nested loops to expose parallelism across iterations [Nic84a]. This is important for good compaction because the parallelism may not be found in the innermost loop. The techniques of the previous section can then be used to compact the resulting loop. Tree height reduction techniques [Kuc76] are very useful in conjunction with quantization. Quantization applies to non-linear as well as linear recurrences; in fact, quantization is limited only by the degree to which indirect references can be disambiguated.

The idea of loop quantization is to unwind a few iterations of all nested loops; the unwindings chosen should minimize the lengths of dependency chains. However, the order of data-dependent statements must not be altered. To quantize n nested loops with loop indices $I_1, I_2, \ldots, I_n$, loop i is unwound k_i times by duplicating the loop body k_i times. In the first duplication of the original body the index I_i is unchanged; in the second each occurrence of I_i is replaced with $I_i + 1$, and so on, up to $I_i - k_i - 1$. This procedure is repeated for each nested loop proceeding from the innermost loop to the outermost loop. This is equivalent to unwinding all the nested loops fully

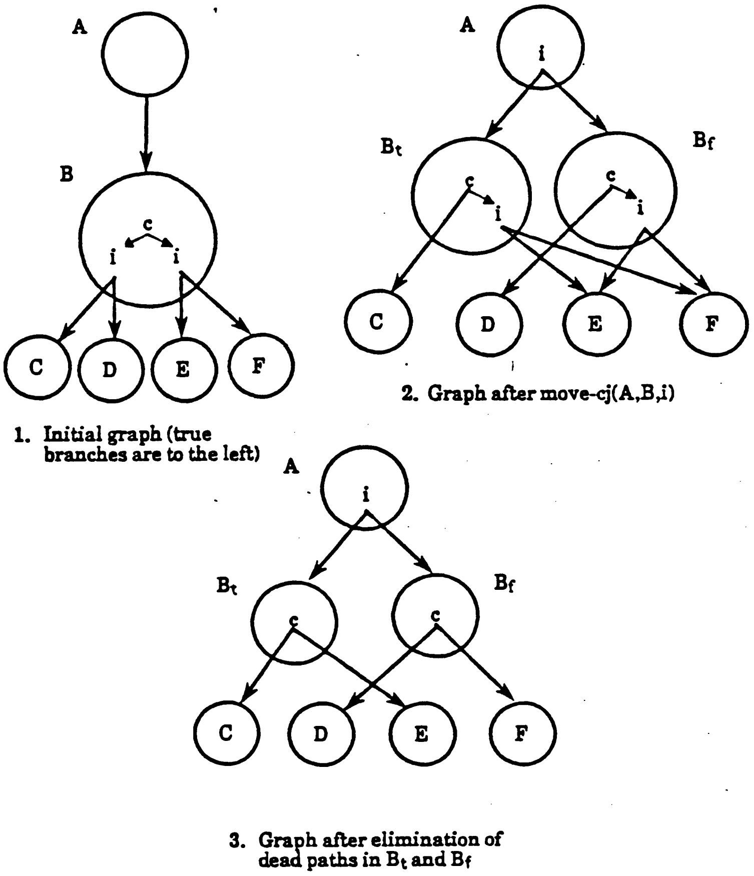


Figure 9: Removing a redundant test.

when the upper bounds are $k_1, \ldots, k_n$.

When multiple loops are unwound, an iteration of the loop executes an *n−dimensional* box B of size $k_1 \times k_2 \times \ldots \times k_n$. All statements in B are executed before the box is shifted by a "quantum jump" along any of the dimensions. The movement along the n dimensions, while quantized, is in normal loop order. The conditions under which a quantization preserves correctness can be found in [Nic84a].

4.3 General Support

The *General Support* level finds and records data dependencies. Memory disambiguation and enhanced flow analysis methods increase the accuracy of data dependencies and permit more code motions [Nic84b,Har77]. Traditional optimizations, such as dead code removal, are also used at this level.

A good example of a general support transformation is *dead path elimination*. Moving a conditional-jump can produce a program graph with some paths that can never be taken. In Fig. 9, 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 [Har77], 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 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.

4.4 The Profiler

The system includes a profiler to provide assistance in the optimization process. The profiler can be used in two modes. In the first mode the program is run on "typical" data, and statistics–such as the frequency of execution of blocks of code and frequency of data dependencies between indirect memory references–are gathered automatically. Alternatively, if such exploratory runs are unrepresentative or too expensive to perform without parallelization an interactive mode can be used. The system then uses micro-analysis [Coh82] to evaluate the time complexity of the program and to identify "hot-spots." As the user improves the code the profiler must dynamically update its estimates. Accurate updating is non-trivial as it involves knowledge of the target machine and its influence on the running time of the compacted code. A simple version of the profiler is currently operational in our system.

5 Current Work

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 this level is a tabular description of the microengine for which code is to be generated. It can be easily modified by the user to match a particular microengine architecture. The description is currently rather simple-minded. In the next stage of our work we plan to use more accurate machine descriptions, drawing on the work of [DWH86,Dam85]. We will also improve the retargetability of the environment and extend the environment's ability to integrate and exploit realistic machine characteristics.

Two major extensions of the environment are currently being included in the *Mapping Layer*: the addition of resource constraints and the elimination of the assumption that all operations require one cycle to execute. The term resources includes everything that is required to execute an operation: bus lines, registers, functional units, etc. An operation is not allowed to move into a node if the node would then require more resources than the machine has available.

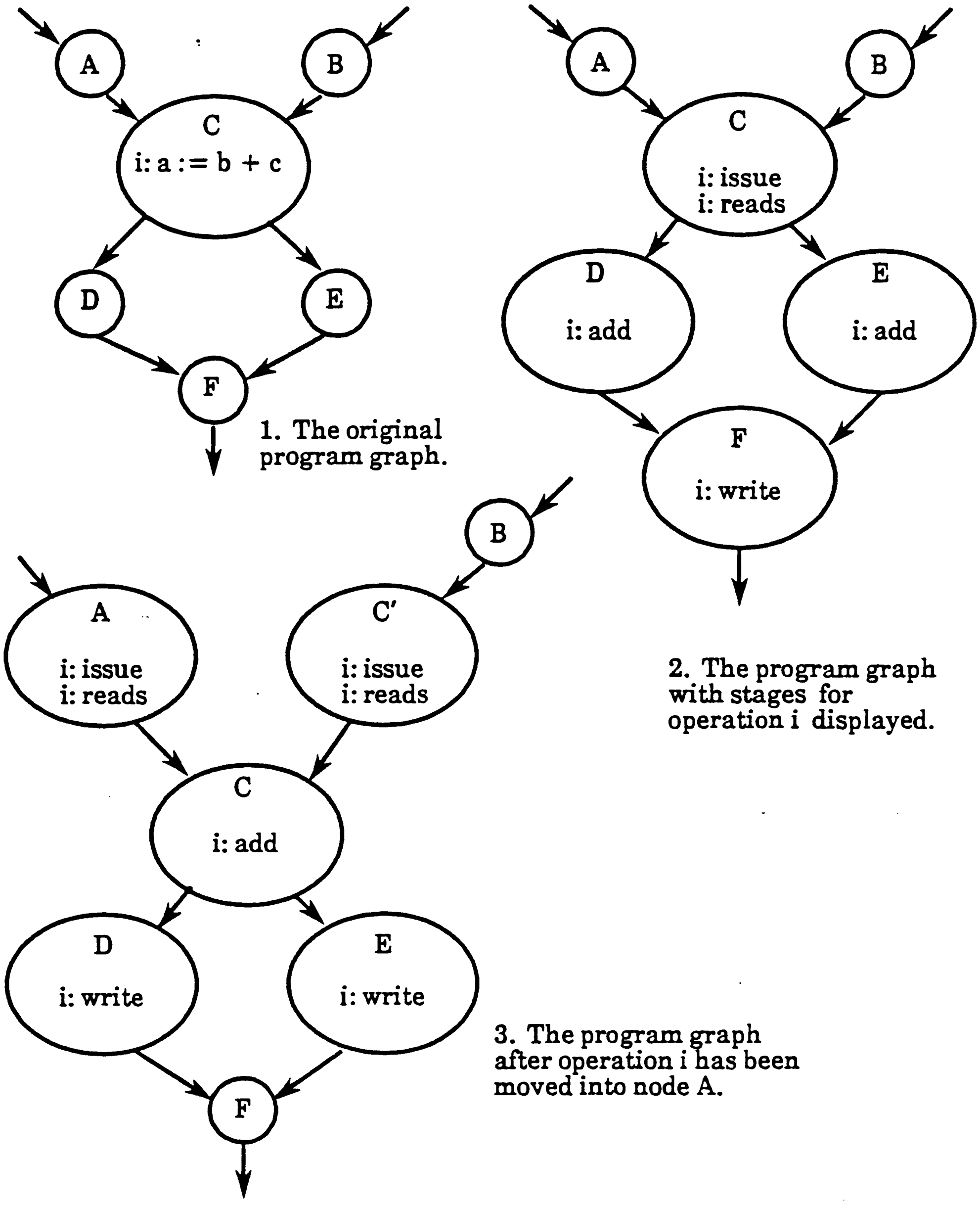
Dealing with variable length operations requires changes in the definitions of the core transformations. Besides the constraints already stated, now an operation may move only when there are no conflicts with operations it overlaps in time.

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. 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. 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. 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. 10.

The *Mapping Layer* will be transparent; the user will still deal with the high-level representation of the program and may rely on the system's heuristics (e.g., list scheduling [Fis77]) to perform the mapping of the program to hardware. However, the user may wish to control resource allocation and operation pipelining directly. To allow for this operations can be expanded to display operation stages in the program graph.

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. In current systems the user must accept suboptimal results or the microcode must be compacted without automatic support. This is, at best, a tedious and error-prone process. The use of an environment to assist in the parallelization process will greatly reduce the time and effort required to optimize a program and lead to better code.



1. The original program graph.

2. The program graph with stages for operation i displayed.

3. The program graph after operation i has been moved into node A.

Figure 10: A pipelining example.

6 Implementation

The primary consideration in implementing the system is that the user's view should be as high-level and abstract as possible. In interacting with the system the user deals only with an abstract model of computation [Nic84c] that provides access to fine-grained parallelism without the burden of architectural, semantics-preservation, and synchronization details.

The environment actually resides on two machines, a Vax 11/780 and a Xerox Dandelion workstation. 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 transformations are executed on the Vax. The 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 an operation, select "Move" from a menu of options, and then click on a destination node. The environment then tries to move the operation to the destination node using the *move-path* transformation.

The high-level source language currently used by the environment is Tinylisp, a Fortran-like language with lisp syntax. The machine-level language is called naddr [FERN84]. Naddr can be easily extended to support typical microcode operations. The choice of language is of no concern to the environment proper; the environment is only concerned with dependencies and flow of control. The model of computation [Nic84c] is rich enough to allow typical microcode 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. This has allowed us to concentrate on the problems of interest - the environment issues and transformations. As our work progresses we plan to convert to other languages better suited for microcode programming [DS78].

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 environment and PS should serve both as a microcode development tool and as a research vehicle for investigating new automatic code transformations. Therefore, experimentation with the system beyond the designers' group is desirable and portability for wide distribution is a main concern.

7 Early Experimental Results

Our environment 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

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

meant to demonstrate the viability of our approach under such adverse conditions as 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 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.

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 in later instructions either automatically or under user guidance. Such changes do not seem to affect the compaction and speedups significantly. 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. We expect our new environment to be functional within the next one to two months.

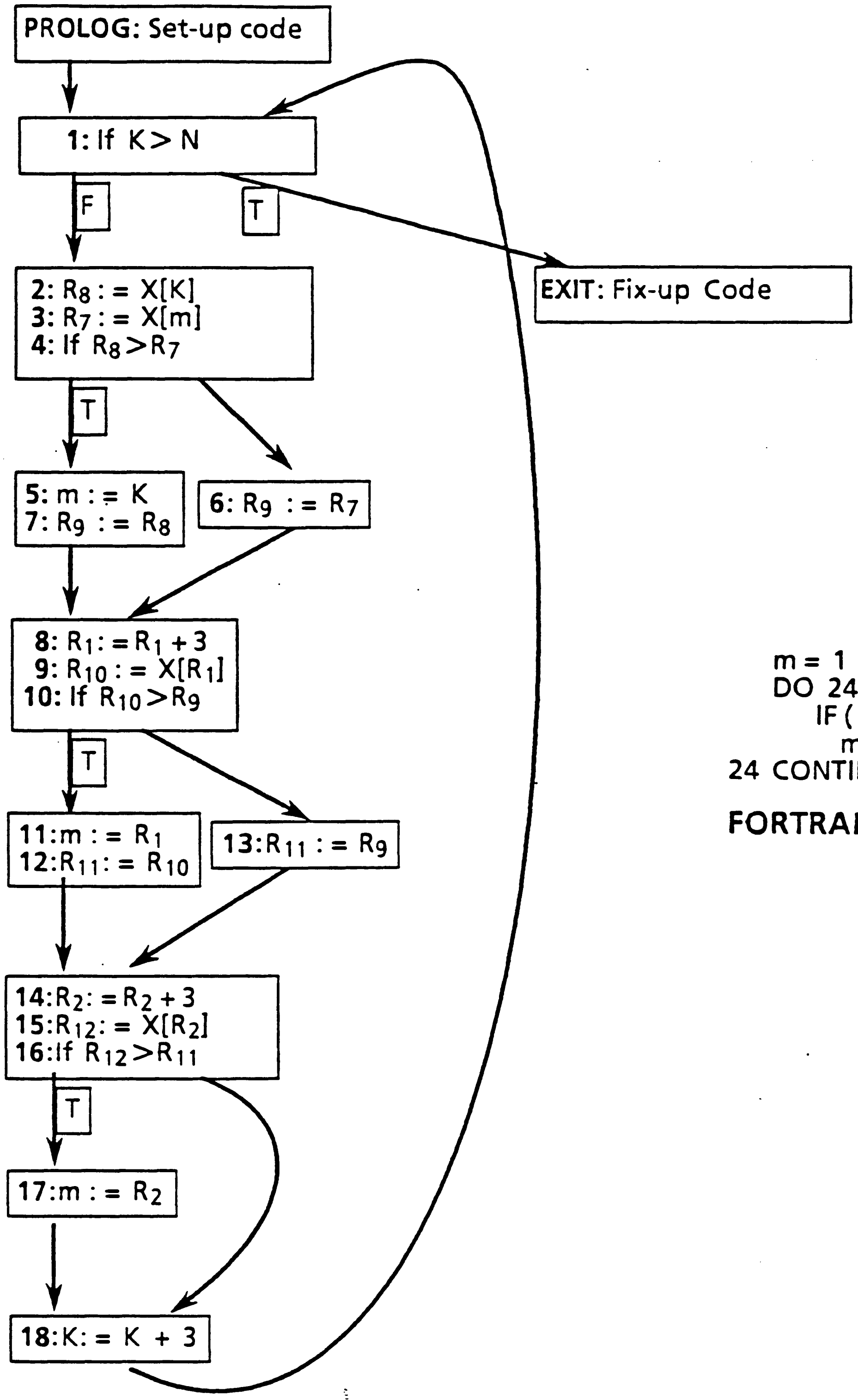
8 Sample Use of the Environment

This section illustrates the transformations achievable in our environment. The transformations involved are relatively simple; their application under user control serves to illustrate a possible mode of interaction with the environment and roughly corresponds to its capabilities to date. 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 simplicity, we assume unit time execution for all operations and no resource conflicts.

Fig. 11 shows the source code for Livermore Loop 24 and the result of unwinding the loop three times. Several standard optimizations (e.g., renaming of index variables, constant-folding) have also been performed by the system.

At this point, the user may specify where the loop is to be broken (*cycle-breaking* transformation). In this example, the user elects to leave the loop body as it is. After one application of *compact-path* most of the compaction has been performed. The path selected consists of al! true branches in the loop body. Fig. 12 shows the result after dead-code removal and a few invocations of *migrate* to bunch conditional-jumps together.

We have omitted many details in this example. For example, *migrate* uses several simple algebraic enabling transformations; in operation 10, R_7 was substituted for R_9 as a result of operation 10 moving above operation 6. Similarly, flow-analysis and peephole optimizations remove redundant memory fetches, while dead-code removal eliminates redundant assignments to m.




```
      m = 1
      DO 24 K = 2, N
         IF ( X[K] .LT. X[m]) THEN
           m = K
24 CONTINUE
```

FORTRAN CODE

Figure 11: Sample original program and partially transformed code.

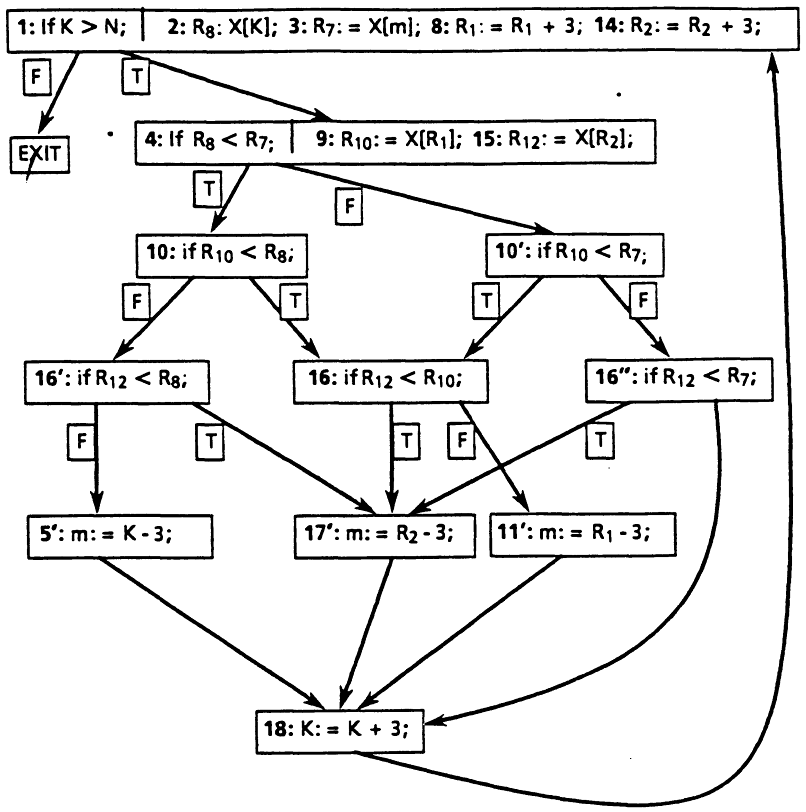


Figure 12: Final transformed code.

Automatic disambiguation of indirect references is successful in removing spurious dependencies in this example; otherwise, some of the motions involving indirect references would appear illegal to the system.

While the transformations were controlled at a relatively low level, the user did not deal with actual hardware. The code in Fig. 12 is an "abstract parallel" schedule. The actual mapping to hardware is done by the system. The user's choice of transformations can be aided by the system's ability to take machine restrictions into account (e.g., instruction times, resource availability). Thus the schedule obtained could map well onto the hardware without the user being intimately familiar with the architecture. As our work progresses, we will integrate higher level transformations into the system. For example, the user will be able to specify code motions in terms of the high-level language statements and constructs.

The speedup for this example is 300% assuming the hardware supports a multiway jump mechanism, or 130% otherwise. If hardware restrictions permitted, this could be further increased by additional unwinding and compaction.

9 Conclusions

We have described a tool for the interactive extraction of fine-grain parallelism from ordinary code. As our work progresses we expect to further automate the parallelization process, by integrating more expertise into the guidance layer of PS and into a machine-specific mapping layer.

References

[AK82] J. R. Allen and K. Kennedy. *PFC: A Program to Convert Fortran to Parallel Form.* Technical Report MASC TR 82-6, Rice University, 1982.

[Coh82] J. Cohen. Computer-assisted microanalysis of programs. *Communications of the ACM*, 25:724–733, October 1982.

[Dam85] W. Damm. Design and specification of microprogrammed computer architectures. In *Proc. 18th Annual Workshop on Microprogramming*, pages 3–11, December 1985.

[DS78] S. Davidson and B. Shriver. An overview of firmware engineering. *IEEE Computer*, 11:21–34, May 1978.

[DWH86] S. Dasgupta, P. A. Wilsey, and J. Heinanen. Axiomatic specification in firmware development systems. *IEEE Transactions on Software*, 3:49–58, July 1986.

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

[Fis77] J. A. Fisher. Trace Scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, C-30:243–50, May 1977.

[GPKK82] D. D. Gajski, D. A. Padua, D. J. Kuck, and R. H. Kuhn. A second opinion on data-flow machines and languages. *IEEE Computer*, 15:58–69, February 1982.

[Har77] W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-3:243–50, May 1977.

[HK84] R. T. Hood and K. Kennedy. *A Programming Environment for Fortran*. Technical Report COMP TR 84-1, Rice University, 1984.

[KKP*81] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proc. ACM Princ. of Prog. Lang.*, pages 207–218, January 1981.

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

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

[MD86] R. A. Mueller and M. R. Duda. Formal methods of microcode verification and synthesis. *IEEE Software*, 3:38–48, July 1986.

[NF84] A. Nicolau and J. Fisher. Measuring the parallelism available for Very Long Instruction Word architectures. *IEEE Transactions on Computers*, C-33:968–76, November 1984.

[Nic84a] A. Nicolau. *Loop Quantization, or Unwinding Done Right.* Technical Report 85-709, Cornell University, 1984.

[Nic84b] A. Nicolau. *Parallelism, Memory Anti-Aliasing and Correctness for Trace Scheduling Compilers.* PhD thesis, Yale University, 1984.

[Nic84c] A. Nicolau. *Percolation Scheduling: A Parallel Compilation Technique.* Technical Report 85-678, Cornell University, 1984.

[PHS85] Y. N. Patt, W. Hwu, and M. C. Shebanow. HPS, a new microarchitecture: rationale and introduction. In *Proc. 18th Annual Workshop on Microprogramming*, pages 103–8, December 1985.

[Sny83] L. Snyder. *Introduction to the Poker Parallel Programming Environment.* Technical Report CSD-TR-432, Purdue University, 1983.

[WC86] R. I. Winner and E. M. Carter. Automated vertical migration to dynamic microcode: an overview and example. *IEEE Software*, 3:6–16, July 1986.

[Weg86] S. R. Wegdahl. Microcode optimization: examples and approaches. *IEEE Software*, 3:59–68, July 1986.