A Development Environment for Scientific Parallel Programs*

Alexandru Nicolau
TR 86-722
December 1985

Department of Computer Science
Cornell University
Ithaca, NY 14853

Abstract

This paper describes a development environment for parallel scientific code. The environment uses Percolation Scheduling, a transformational system for parallelism extraction, and an interactive profiling system to give the user control over the parallelization process while reducing the burdensome details of architecture, correctness-preservation and synchronization. Through a graphical interface the user suggests what should be done 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.

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

1 Introduction

This paper describes a development environment for parallel programs. The environment consists of a system of parallelizing transformations, an interactive profiler and a graphical interface. The components of the system, their use, and current and future work will be discussed.

The environment is geared towards mapping scientific programs onto massively-parallel computers. It is envisioned as part of a scientist's workstation, to serve as a front end for the NSF Supercomputing Center at Cornell. The system will have two front-ends: Fortran, for compatibility and ease of transition, and eventually GIBBS, a very high-level language for scientific programming currently under development at Cornell [6]. A first version of our environment will generate code for the current IBM/FPS-264 Production Supercomputer. A later version will generate code for the Massively Parallel Supercomputer that the Center will purchase early next fall.

Existing compilers for parallel machines do not provide the needed support for exploiting parallelism. Important advances in parallelizing ordinary code have been achieved [1], [5], [10]. However, code is still hand-optimized when speed is critical (e.g., [3]). Interesting work has been done in the development of environments for supporting parallel computation [7], [15]. This work has dealt with coarse parallelism and has provided support in configuring pre-optimized modules into coherent concurrent systems. By contrast, we are interested in semi-automatically extracting fine and coarse grained parallelism in a uniform environment in which the user provides suggestions (direction), while the system attempts to instanciate them using semantics-preserving transformations. Our system preserves the semantics of the original program throughout the parallelization process while automating the analysis and the mundane aspects of parallelization. The system will also incorporate expert knowledge about the specific parallel machine for which code is generated, freeing the user of the necessity 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, without the help of experts and in much shorter time.[1]

[1]The ability of the system to keep track of the complex low-level details (e.g. data-dependencies and architectural details) may give it an edge over expert human users.

At the heart of our environment is Percolation Scheduling (PS). PS is a system of semantics-preserving transformations that convert an original program graph into a more parallel one. PS globally rearranges code in an attempt to gain parallelism. Its core consists of a small set of primitive program transformations defined in terms of adjacent nodes in a *program graph*. These transformations are easy to understand and implement. Furthermore, they 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. 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.

2 Architectures

Several existing architectures could benefit from the use of our environment. Even non-parallel architectures such as lookahead (pipelined) machines could benefit by simply using the large numbers of sequential operations clustered together by percolation scheduling to efficiently fill long pipelines. Furthermore, since through the environment/compiler most of the work could be done statically at compile time, there would be little need for (cumbersome and expensive) hardware synchronization mechanisms and pipe flushing customarily used. 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 [12].

Also 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 [8]. We believe that the environment described in this paper could be put to good use in this context. Through PS transformations a correct *partial ordering* for the issuing of operations can be obtained at compile time and a reasonable partitioning of the program and data between the various processors could be achieved. 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.

Multiprocessors could also benefit from the use of the proposed environment and techniques. Here, too, simple hardware support for parallel conditional jump evaluation could help to take advantage of the parallelism exposed by PS.

3 The Core of PS

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 that together form Percolation Scheduling. Higher levels of this hierarchy direct the core transformations and rearrange the program graph to allow more code motion by the core transformations. Aided by the other levels, the core transformations operate uniformly and can be applied to partially parallelized code, allowing PS to improve code produced by other compilers.

The following is an overview of the PS hierarchy and the work we have completed. A formal description of the model of computation as well as proofs of correctness of the core transformations can be found in [14].

3.1 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 operations to "percolate" (move towards the top of the program-graph) from the various parts of the program graph towards the start node, hence the name Percolation Scheduling.

The details of the transformations deal with maintaining the integrity of all affected paths. A brief description of each transformation is given below. Rigorous definitions can be found in [14].

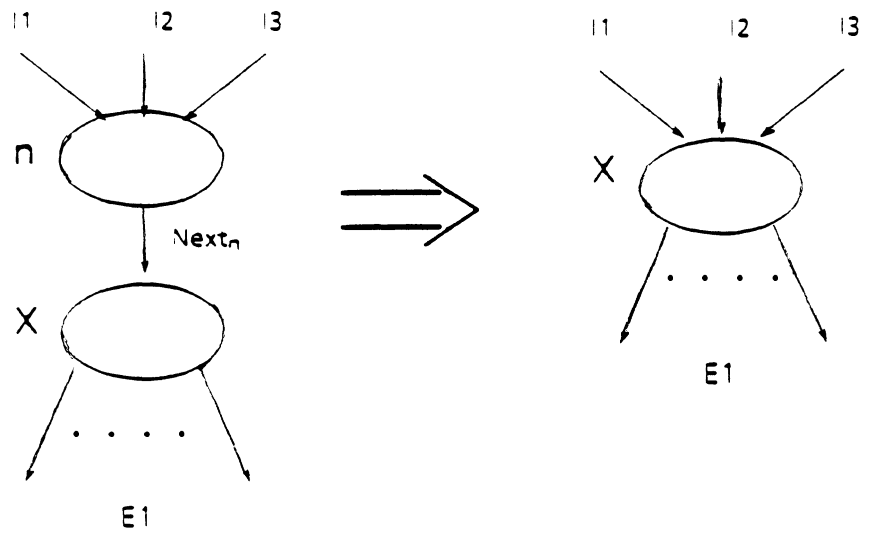

Figure 1: Delete Transformation

3.2 Delete Transformation

A node in the program graph can be removed by the *Delete* transformation when the node has no components (i.e., it contains no executable operations). 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 outgoing edges of their predecessors are reset to point to the successor of the deleted node. This will preserve the semantics of the original program. An illustration is given in Figure 1.

3.3 Move-op Transformation

This transformation moves an operation that does not affect the control-flow up from node n to node m, through edge (m, n), provided no data-dependency exists between operations in m and the operation being moved. In performing the movement, care must be taken not to affect the semantics of paths passing only through n but not through m. To ensure this, these paths are split and provided with a copy of the original n. An illustration is given in Figure 2.

3.4 Move-cj Transformation

This transformation moves a conditional-jump operation up 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

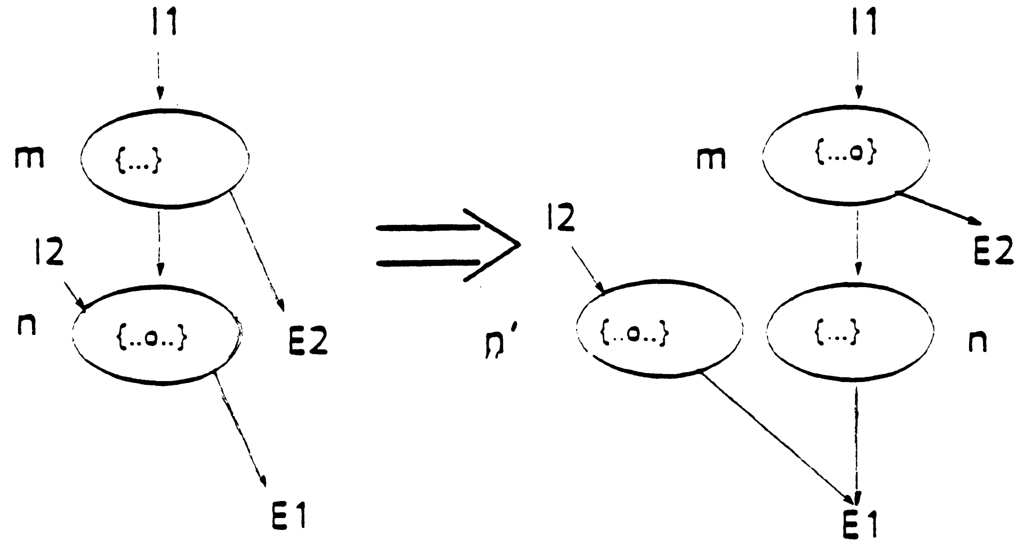

Figure 2: Move-op Transformation

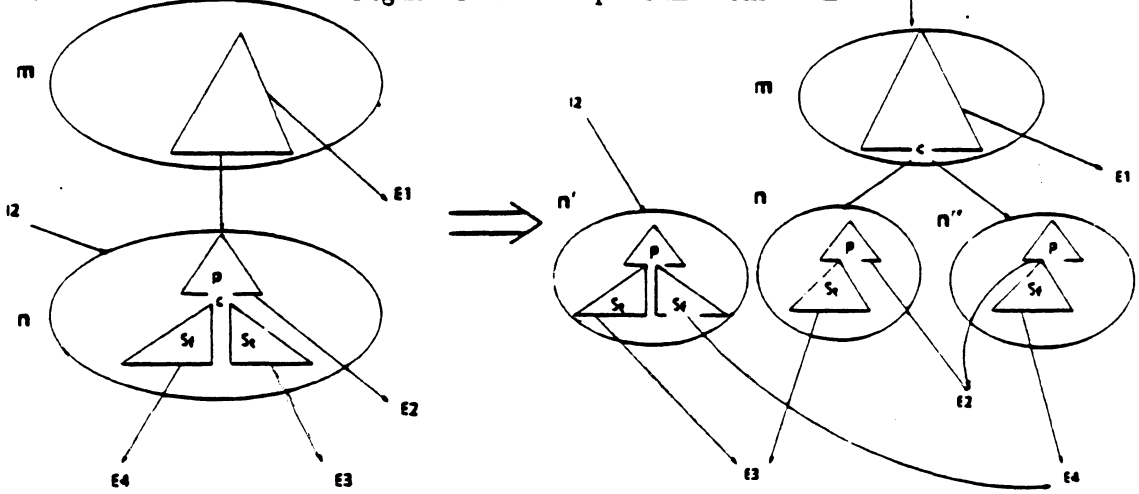

Figure 3: Move-cj Transformation

through n but not through m. To ensure this, the paths are split and a copy of n (called n' in Figure 3) is provided. Since *we allow an arbitrary tree of conditional-jumps in a node, and the conditional-jump being moved may come from an arbitrary spot in that tree*, n will be split into n and n'', to correspond to the true and false branches of the moving conditional-jump (see Figure 3). The details of the splitting and a proof that the transformation indeed preserves the semantic correctness of the original program is beyond the scope of this paper and can be found together with proofs of correctness and termination in [14]. A detailed description of a hardware mechanism that efficiently implements general conditional-jump trees of the type supported by PS is found in [12]. An illustration is given in Figure 3.

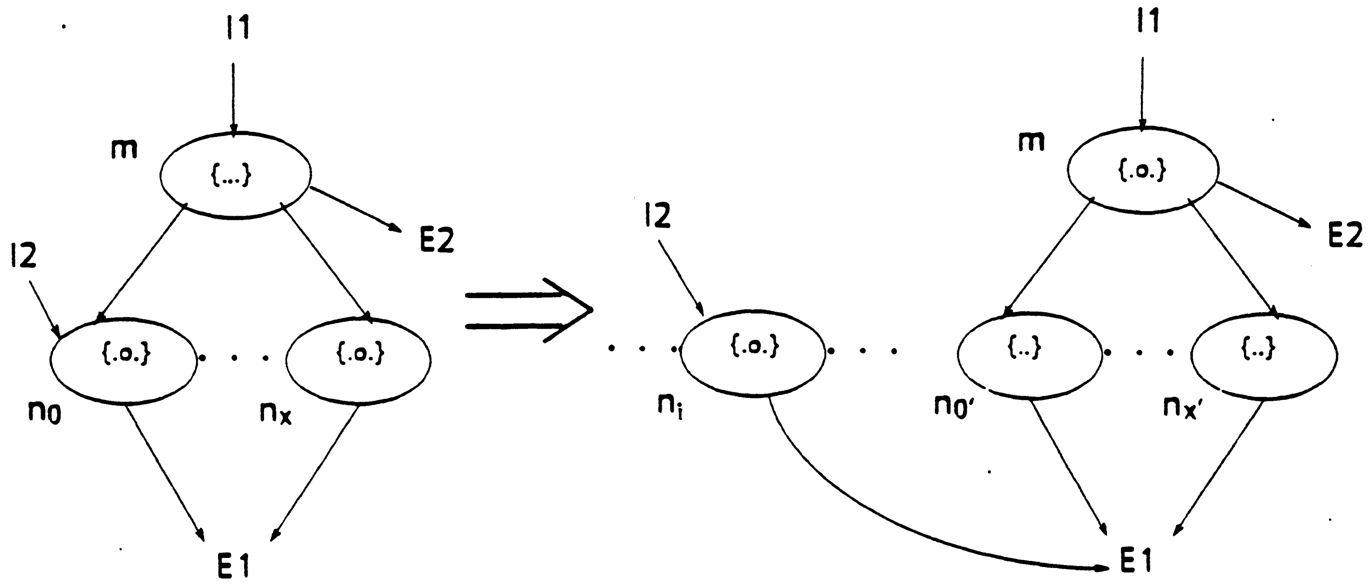

Figure 4: Unification Transformation

3.5 Unification Transformation

This 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 paths (m, n_i) exist for all nodes in the set. In performing the code motion. care must be taken not to affect paths going only through the n_i's but not through m and, as usual, splitting and copying is used. An illustration is given in Figure 4.

4 Beyond the Core Transformations

The core transformations do most of the code motion, and higher levels direct the core transformations and rearrange the program graph to allow more code motion by the core transformations.

General Support: At this level, the data dependencies are found and recorded. Memory disambiguation and enhanced flow analysis methods [11] increase the accuracy of data dependencies and permit more code motions. Traditional optimizations, such as dead code removal, are also used at this level.

Guidance Rules: This level consists of a set of rules that decide when and where to apply the core transformations. Operation and branch probability of execution (Node-probability rule), reverse depth-first ordering, dependency-chains lengths and many other heuristics [14] can be used to direct the core transformations, by assigning priorities to the transformations and directing their application to the "important" parts of the code first. Because this layer is completely independent from the core transformations, we have the freedom of chosing a wide variety of heuristics that can be easily changed for particular applications.

Enabling Transformations: This level provides transformations of the program graph that cannot be accomplished by the simple code motions of the core transformations. The program graph rearrangements are done primarily to allow the core transformations greater freedom to move code. For example, the Cycle-Breaking transformation (conceptually) breaks cycles in program graphs, enabling the application of the core transformations to loops. The point where the cycles are broken is chosen so as to minimize dependencies in the modified loop-body. As a result, the loop may be significantly rearranged. Other examples of such rearrangements are: folding, tree-height reduction [9], and loop quantization, a multiple loop unrolling technique [13].

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 Delete and Move-op transformations can be applied to meta-nodes. Components of the meta-nodes are allowed to percolate, subject to its aggregate data-dependencies. We have developed several techniques that allow the static overlapping and synchronization of independent, or partially independent loops, even in cases where the loops are not vectorizable (see section 4). For example overlapping of loops/modules can be statically achieved at compile time, even when the loops/modules are not data-independent.

Mapping Percolation Schedules to Hardware

The transformations given above expose the parallelism available in the code and provide a partial ordering on the issuing of the operations. The transformed program 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. To execute the resulting code on realistic architectures, we need a mechanism to change the ideal schedule into a schedule that respects resource limitations. Even for simple architectural models, finding an optimal schedule is NP-hard. Fortunately, greedy algorithms can be used to obtain good mappings. For PS we use a refinement of List Scheduling, which is reportedly very effective [5], [4].

5 The Environment Proper

Our ultimate goal is to have a PS compiler generate better code than human experts. However, due to the complexity of the code-generation problems, the compiler must rely on heuristics, which may sometimes fail to produce optimal or close-to-optimal code. Furthermore, the human user will usually have at his disposal a vast amount of knowledge about the problem he is attempting to code and thus may be able to make decisions based on information not available to the compiler. For example, the knowledge that a certain variable can only take positive values may be obvious to the programmer because he knows that the variable stands for a certain physical quantity (e.g., the rest mass of a particle), without that information being explicit in the actual code. The support environment we are building is designed to allow the user to control the parallelization process and provide an integrated interface through which to supply additional information that may help in the optimization of the code.

A PS based compiler will be integrated in our environment for parallel-program development. The role of the compiler is to extract most of the easily achievable parallelism. While this may sometimes suffice, the user may wish to fine-tune his code for better performance. The other components of the system, the profiler and the graphical interface, are being designed to support this activity.

The profiler can be used in two modes. First, the program can be run on "typical" data and statistics (e.g., frequency of execution of blocks of code, frequency of data-dependencies between indirect memory references, and structure sizes) can be gathered automatically. Alternatively, if such exploratory runs are unrepresentative or too expensive to run without parallelization, an interactive mode can be used. The system then uses micro-analysis [2] to evaluate the complexity (running time) of the program and to identify "hot-spots" with the user (or the system, for some obvious situations) supplying estimates of the above statistics. The user can then concentrate his efforts on improving the bottlenecks first. The profiler must dynamically update its estimates as the code becomes more parallel. Accurate updating is non-trivial, as it involves knowledge of the parallel architecture and its influence on the time complexity of the parallelized code. However, accurate information is vital in achieving good parallelization. Without such information, one may optimize unimportant code first, possibly inhibiting later optimization of major bottlenecks. An additional tool that can provide insight into the optimization process is the dependency-chain analyzer. This module displays data-dependency chains between nodes in the program graph, based on the information currently available to the system. The dependencies between the nodes are weighted by the execution-times of the nodes, so that the chains accurately reflect execution times. Dependencies may be altered (clarified) by the user who may supply additional information.

Using the information obtained by the profiler and the dependency chain analyzer, the user can proceed to improve his code. The user provides the insight for what will be usefully done in parallel, while the system performs the actual changes, through semantics-preserving transformations provided by PS. A sequence of transformations that satisfies the user request is automatically determined by the system, with the help of simple built-in guidance rules. In case a request cannot be satisfied, the system will report the problem encountered and ask for help. The user may resolve the problem or decide that the problem occurs infrequently. Such information may be available to the user in the problem domain, but may not appear explicitly in the program, so that it could not be inferred by the system. Alternatively, the code may be changed to eliminate the problem.

In interacting with the system the user only deals with the high-level language and with an abstract model of parallel computation [14] that provides access to fine-grained parallelism, without the burden of architectural, semantics-preservation, and synchronization details. The interface to the system is graphical. Through windows, different views of the code may be obtained. For example, the program can be displayed in source or abstract parallel program-graph form, and the user can choose to "zoom" in or out on parts of the code, displaying more or less of the parallelization details. As the user "focuses" on some part of the code, the data-dependency graph and the time-complexity information for that piece of code can be displayed, again at various levels of abstraction. The user requests can be indicated graphically, by pointing to and selecting parts of the code and indicating desired moves or other operations.

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. In this context Expert Systems may prove helpful. Expertise is required in dealing with complex user requests and architectures, without extensive backtracking (from situations that do not satisfy the request) and without often resorting to user help.

To date, we have completed the first part of this research. We have formally defined the core PS transformations, established their semantic correctness and identified and defined guidance rules and transformations of the higher layers. We have also obtained encouraging preliminary evidence of the effectiveness of PS. An implementation of a version of the environment for a hypothetical, very parallel machine is now well underway. The system is written in LISP to facilitate cooperation with tools such as MAXIMA and to allow the integration of Expert Systems techniques into our program development environment.

6 Sample Use of the Environment

This section illustrates the transformations achievable in our environment. The transformations involved are relatively simple, and could be guided by the simple heuristics described previously. Their application under user control serves just 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 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 [12] and larger programs.

Consider the sample program in figure 5 (Livermore Loop 24). Figure 6 shows the result of unwinding the loop three times (achieved by the "unwind" command). Several standard optimizations (e.g., renaming of index variables, constant-folding) have also been performed by the system at this point. The user may then specify the point at which the loop is to be broken (Cyclic Motion rule) and allow the system to perform the core transformations guided by the built in heuristics. In choosing the break-point of the loop, the user may rely on information displayed by the profiler and dependency-chain analyzer. Alternatively, the (core) transformations may be guided "manually", by selecting the operation which should move, its destination in the program-graph and the direction in which the move should take place. In the case of a loop, operations may be moved "forward" in the loop-body or "backward" through the back-edge of the loop.

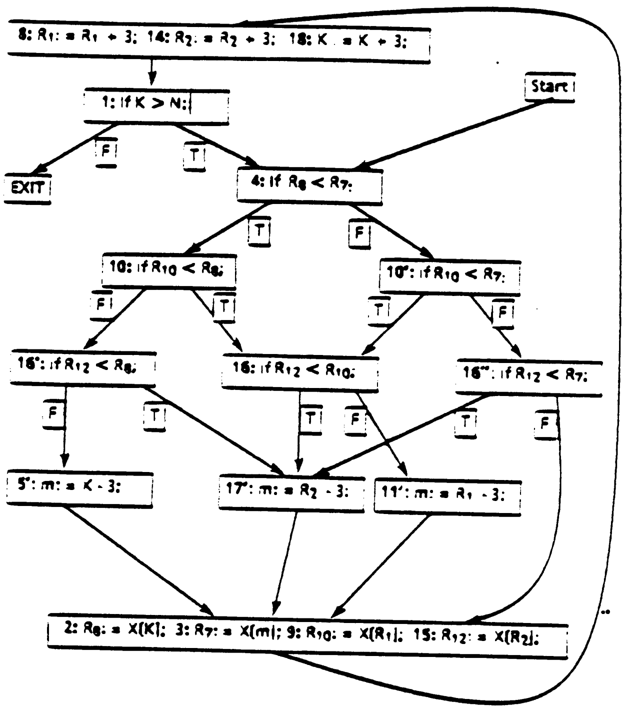

Figure 7: Final transformed code.

Next the core transformations are applied, subject to the user's directions and/or the Guidance Rules Level and the General Support Level of PS. Automatic disambiguation of indirect references is successful in removing spurious dependencies in this example; otherwise, some of the motions involving indirect references will have appeared illegal to the system who would have complained and requested help.

The resulting code is shown in figure 7. In the process, several simple algebraic manipulations can be performed by the system to support the application of the code transformations. For example, operations of the form $m := K$ are changed to $m := K - 3$ as a side-effect of allowing $K := K + 3$ to percolate upward. This may or may not help increase the effectiveness of the optimization, (i.e., lead to a shorter schedule) depending on the particular architecture for which we optimize. Similarly, flow-analysis and peephole optimizations remove redundant memory fetches, while dead-code removal eliminates redundant assignments to m. Due to the small size of the example, some of the higher level transformations (e.g., Meta-transformations) are not applicable.

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

FORTRAN CODE

Figure 5: Sample original program.

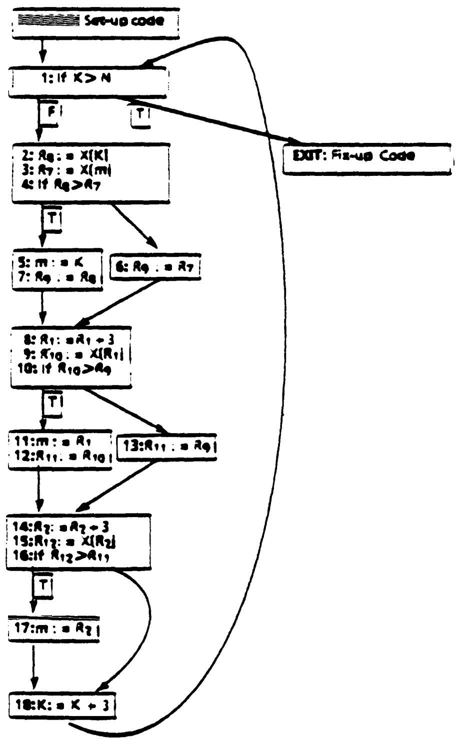

Figure 6: Partially transformed code.

While the transformations were controlled at a relatively low level, the user did not deal with actual hardware. The code in figure 7 is just an "abstract parallel" schedule. The actual mapping to hardware will be done by the system. Nevertheless, the user's choices can be "guided" by the system, since PS transformations, the profiler, and the dependency-chain analyzer may be forced to take into account machine restrictions (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.

7 Acknowledgements

The comments of my colleagues Ken Birman, Alan Demers, David Gries, Kevin Karplus, Fred Schneider and Jon Solworth are gratefully acknowledged.

References

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

[2] J.Cohen. *Computer-assisted microanalysis of programs*. Communications of the ACM 25, 10, pp.724-733, 1982.

[3] 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.

[4] 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.

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

[6] The Gibbs Group. *GIBBS—A Programming Environment and Workstation for Scientists.* Center for Theory and Simulation in Science and Engineering Technical Report CTSSE 84-7, 1984.

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

[8] 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.

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

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

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

[12] A.Nicolau and K.Karplus.*ROPE: A Statically Scheduled Supercomputer Architecture.* (Submitted to the First International Conference on Supercomputer Systems, December 16-20, 1985).

[13] A.Nicolau. *Loop Quantization, or Unwinding Done Right.* Cornell University, Dept. of Computer Science Technical Report, 1984.

[14] A.Nicolau. *Percolation Scheduling: A Parallel Compilation Technique.* Cornell University, Dept. of Computer Science Technical Report, 1984.

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