A Development Environment for Horizontal Microcode Programs

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 to give 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 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.

1 Introduction

This paper describes a development environment for microprograms. 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.

Our ultimate goal is to have a compiler generate better code than human experts. However, due to the complexity of the code-generation problems, the compiler must rely on heuristics which sometimes fail to produce optimal or nearly 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, it may be obvious to the programmer that a certain variable can only take positive values without that information being explicit in the actual code. The support environment we are building is designed to allow the user to control the compaction process 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 extract most of the easily achievable parallelism. While this may sometimes suffice, the user may wish to fine-tune the code for better performance. The other components of the system, the profiler and graphical interface, are being designed to support this activity.

The environment is geared towards mapping 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.

*This work is supported in part by the Cornell NSF Supercomputing Center.

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

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 [1], [4], [9]. Interesting work has also been done in the development of environments for supporting parallel computation [5],[15]. 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, expert humans are generally much better at microcode compaction than any available system. 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 in which the user provides directions which the system attempts to instantiate. 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 in much less time.[1]

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[3]. PS is a system 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 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 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 the use of our environment. Horizontal microengines are the most obvious. Even non-parallel architectures such as vertical lookahead (pipelined) machines could benefit by simply using 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 exploited by Percolation Scheduling. The design of such an architecture and its advantages are described in [7].

Also suited to take advantage of our system are data-flow microengines [14]. Traditionally, it has been claimed that dataflow architectures require very little compile-time analysis. 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

[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 working without such support.

of memory and resources [6]. 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 functional units 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.

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 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. In these sections, the term *node* (in a program graph) refers to a microinstruction. An operation is a component of some microinstruction. In our examples, we use lower case letters to denote operations and capital letters for nodes. A formal description of the model of computation as well as proofs of correctness of the core transformations can be found in [13].

Four primitive transformations form the core of PS. They are defined in terms of 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. As PS is used on a program graph, operations will be packed together in nodes, 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. Rigorous definitions can be found in [13].

3.1 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.

3.2 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

Figure 1: Delete Transformation

Figure 2: Move-op Transformation

computation 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.

3.3 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 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 rooted DAG of conditional-jumps in a node, and the conditional-jump being moved may come from an arbitrary spot in that DAG,* N will be split into N and N'', to correspond to the true and false branches of the moving conditional-jump. 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 [13]. An illustration of the transformation is given in figure 3. A detailed description of a hardware mechanism that efficiently implements general conditional-jump DAGs of the type supported by PS is found in [7]. While a multiway jump mechanism will take full advantage of the power of PS, it is not required for the use of our system. Our environment can be used to generate good code for any horizontal architecture.

Figure 3: Move-cj Transformation

Figure 4: Unification Transformation

3.4 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 through the N_i's but not through M - as usual, splitting and copying is used. An illustration is given in figure 4.

3.5 Inverse Transformations

Functional 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 useful in situations where it is necessary to undo a transformation. Such situations arise because an operation i can "percolate" to a node where it prevents another operation j from moving that could otherwise move. If at some point it would be more advantageous to move j rather than i then i must be moved (via 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 necessary 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; however, we would like to maintain as much uniformity in the system as possible. Our solution has been to group our 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 by the move. If a potential data dependency violation is discovered, the operation is moved as far as possible and the conflict is reported to the user. The only other restriction on *move-path* is that the path between the two instructions be unique.

A more powerful transformation is *migrate*. *Migrate* takes an operation and a node as its arguments, and then moves the 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, in figure 5, assume that statements i and j 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 even be moved into nodes A and C unless unifications are performed because of data dependency conflicts with other copies of j.

Let i denote the operation we wish to move, and let $\mathrm{I}(t)$ denote the set of all copies of i in the program graph at time t. We define the function $\mathrm{node}(j)$ 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 $\mathrm{I}(t)$ such that j can be moved from $\mathrm{node}(j)$ to any predecessor of $\mathrm{node}(j)$.

2. Let $\mathrm{reach}(Y)$ denote the graph of all nodes reachable from a node Y. If j is moved from $\mathrm{node}(j)$ to some predecessor X of $\mathrm{node}(j)$ at time t, then the operation is a unification and there is no k in $\mathrm{I}(t)$ such that $\mathrm{node}(k)$ is in $\mathrm{reach}(X)$ and k can move to some predecessor of $\mathrm{node}(k)$ in $\mathrm{reach}(X)$.

Figure 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. The problem with loops is that 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 backedge. 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 first 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 into L. If i is loop invariant with respect to L it is removed from L and inserted immediately before the entry point to L. If i is not loop invariant, then it is not permitted to move into L.

Figure 5: Unification Example

Figure 6: Migrate Transformation

The first condition prevents operations initially in a loop body from being moved indefinitely around the backedge of the loop. We assume 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 attempt to 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 transformations to the entire program graph. Instead, critical sections of code can be optimized, or simply transformed to expose parallelism, and then global heuristics can be applied to the graph to obtain a good schedule.

Compact-blocks is one such strategy that we have implemented. 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 speed-up achievable by exploiting parallelism within basic blocks is small [11]. In fact, *compact-blocks* is intended for use primarily as a first step in the compaction process. Its application can reduce the number of nodes in the graph considerably without moving any operation to a point where it blocks (due to a data dependency) an operation that would otherwise have been able to move.

Another global heuristic is *compact-path. Compact-path* only moves operations on the "most-important" path in a program. We assume that for each conditional j we have two real numbers true(j) and false(j) representing the probabilities that the true branch of j and the false branch of j will be followed respectively. In many cases such analysis can be performed automatically with good results [11]. 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 for the edges on the path. In the algorithm for *compact-path* (see figure 7) p(X,Y) denotes the probability that program execution will continue with node Y after the execution of node X.

Figure 7: Compact-path Transformation

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

In our system the user has the option of selecting the path for *compact-path*. For a typical program, we envision that the user would first explicitly select the critical paths through the code for optimization. The system could then select and compact less important paths automatically.

It is interesting to note that *compact-path* alone is strictly more powerful than the technique of trace-scheduling developed in the ELI project at Yale [4]. Trace-scheduling also selects the "important" path (or "trace") through the program and compacts it. However, trace-scheduling does not perform unifications. Figure 8 provides an (admittedly trivial) example for which trying to compact a single trace without unification results in no improvement. Another advantage 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 paths 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 be subsequently unified, thus limiting the size of the final code. This can be a major advantage in microcode compaction when the size of the available microstore is small.

4.2 Enabling Transformations

This level provides transformations of the program graph that rely on global information and thus could not be accomplished by the core transformations. The purpose of these transformations is

Figure 9: An example of variable renaming

primarily to expose parallelism that will subsequently be exploited by the core transformations.

A typical example of an enabling transformation is variable renaming. The judicious choice of new variable names can often remove dependencies between statements. In 9, renaming variable a in operations k and l allows the core transformations to compact this section of code.

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 so that e becomes the backedge (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 multiple loops to expose parallelism [12]. Quantization is used to expose parallelism across several iterations of nested loops. This is important for good compaction, since 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 [8] 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 basic idea of loop quantization is to unwind a few iterations of all nested loops. The unwindings chosen should maximize parallelism exposed (i.e., minimize the lengths of dependency chains). However, care must be taken not to alter the order of data-dependent statements. 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 to the outermost. This is equivalent to unwinding all the nested loops fully when their upper bounds are $k_1, \ldots, k_n$.

When multiple loops are unwound, an iteration of the loop involves executing 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 [12].

We have recently discovered and implemented an algorithm which solves the following quantization problem: given a set of n nested loops, what is the greatest unrolling of every loop for which correctness is preserved and all unrolled iterations are independent? While this will not in general be the best quantization, the computation can be done without any explicit unrolling, thus avoiding the problem of code explosion. Because the iterations in the quantized loop are independent, we can represent the final loop as the original loop body and a vector $< i_1, i_2, \ldots, i_n >$, where i_j is the number of times loop j is unwound.

4.3 Other Levels

In the *General Support* layer data dependencies are found and recorded. Memory disambiguation and enhanced flow analysis methods [10] increase the accuracy of data dependencies and permit more code motions. Traditional optimizations, such as dead code removal, are also used at this level.

A profiler is being incorporated into the system to provide assistance in the optimization process. The profiler can be used in two modes. First, the program can be run on "typical" data and statistics (e.g., the 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 perform without parallelization, an interactive mode can be used. The system then uses micro-analysis [2] to evaluate the time complexity of the program and to identify "hot-spots" with the user (or the system, for some obvious situations) supplying estimates of the above statistics. 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.

The transformations described above expose the parallelism available and provide a partial ordering 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. To execute the resulting code on realistic architectures, we need a mechanism to convert the ideal schedule to a schedule that recognizes resource limitations. Unfortunately, even for simple architectural models finding 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 description of the microengine for which code is to be generated. The description is stored in tabular form, and can be easily modified by the user to match a particular microengine architecture.

The two major extensions of the environment that are included in the *Mapping Layer* are the addition of resource constraints and the elimination of the assumption that all operations require one cycle to execute. Under the heading of resources falls anything that is required to execute an operation: bus lines, registers, functional units, etc. An operation is not allowed to move into a node if adding that operation would result in the node requiring more resources than the machine has available to it.

Dealing with variable length operations requires changes in the definitions of the core trans-

Figure 10: A pipelining example.

formations. Besides the constraints already stated, now an operation i can only move when there are no conflicts with operations it overlaps in time. This is accomplished by searching a region in the neighborhood of the affected node for operations that may conflict with i. We store the structure of the life-cycle for each type of operation - the number of cycles after issue required before loads, arithmetic operations, writes, etc., can be performed. Let m be the length (in cycles) of the longest operation. When operation i is moved, the structure information is used to compare i with all other operations within m cycles on all paths reachable from and reaching to node(i). If any stage of another operation overlaps a stage of i in such a way that data dependencies are violated, the transformation is not performed. An example of *move-op* with stages of the instructions displayed is given in figure 10.

The *Mapping Layer* will be transparent; the user will still deal with the high-level representation of the program, and may rely on heuristics (e.g., list scheduling [4]) built into the system 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 in the environment to explicitly display operation stages in the program graph.

Including resource constraints and variable length operations greatly complicates the conditions under which a transformation is allowed. Fortunately, these checks can be done automatically and efficiently. In current systems, the user has to either accept suboptimal results or the microcode compaction has to be done by hand with no automatic support. This is, at best, a tedious and error-prone process. The use of an environment to assist in the parallelization process would greatly reduce the time and effort required to optimize a program and, and probably lead to better code.

5 Implementation

The guiding consideration in implementing the system was that the user's view should be kept as high-level and abstract as possible. In interacting with the system the user only deals with an abstract model of computation [13] 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. All of the code is written either in Franz Lisp (on the Vax) or Interlisp (on the Dandelion). Lisp was selected for its robust environment of system functions and debugging tools. The initial compilation and all transformations are executed on the Vax. The workstation serves primarily to display the program graph and profiler information and to accept user input.

The system was placed on two machines so that we could take advantage of the graphics capabilities of the Dandelion. Nearly all user commands are issued with the Dandelion's mouse. A typical sequence might be to click on an operation, select "Move" from a menu of options, and then click on a destination node. The environment would then try to move the operation to the destination node using the *move-path* transformation.

6 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 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 simplicity, we assume unit time execution for all operations and no resource conflicts.

Consider the sample program in figure 11 (Livermore Loop 24). Figure 12 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 (*cycle-breaking* transformation). In this example, the user elects to leave the loop body as it is. After one application of *compact-path* (the path consists of all true branches) most of the compaction has been performed. At this point all that remains to be done is dead-code removal and a few invocations of *migrate* to bunch conditional jumps together.

The resulting code is shown in figure 13. We have omitted many details in this example. For example, *migrate* has used 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 removed redundant memory fetches, while dead-code

.

Figure 11: Sample original program.

.

Figure 12: Partially transformed code.

Figure 13: Final transformed code.

removal eliminated redundant assignments to m. Automatic disambiguation of indirect references was successful in removing spurious dependencies in this example; otherwise, some of the motions involving indirect references would have appeared illegal to the system which would have complained and requested help.

While the transformations were controlled at a relatively low level, the user did not deal with actual hardware. The code in figure 13 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.

The speedup achieved even for this simple example is a factor of 4 assuming the hardware supports a multiway jump mechanism, or 2.3 otherwise. If hardware restrictions permitted, this could be further increased by additional unwinding and compaction.

7 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

[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, 00. 724-733, (1982)

[3] 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).

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

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

[6] 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).

[7] K.Karplus and A.Nicolau. *Getting High Performance with Slow Memory.* Proceedings of the 18th Annual Workshop on Microprogramming, Asilomar, CA, December 1985.

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

[9] D.Kuck, R.Kuhn, D.Padua, B.Leasure and M.Wolfe. *Dependence Graphs and Compiler Optimizations.* Proceedings of POPL 81, ACM, pp. 207-218 (1981).

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

[11] A.Nicolau and J.Fisher. *Measuring the Parallelism Available for Very Long Instruction Word Architectures* IEEE Transactions on Computers, Number 11, pp. 968-76 1984.

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

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

[14] Y.N.Patt, W.Hwu, and M.C.Shebanow. *HPS, A New Microarchitecture: Rationale and Introduction.* Proceedings of the 18th Annual Workshop on Microprogramming, Asilomar, Ca, December 1985.

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