Disambiguation, Correctness and Flow-Analysis Issues for Trace Scheduling Compilers

Alexandru Nicolau

TR 84-627
August 1984

Department of Computer Science
Cornell University
Ithaca, New York 14853

Disambiguation, Correctness and Flow-Analysis Issues for Trace Scheduling Compilers

Alexandru Nicolau

Computer Science Department

Cornell University

Ithaca, NY 14853

Abstract

Trace scheduling is a global compaction technique for transforming sequential programs into parallel code. When this investigation began, trace scheduling was unimplemented and many serious questions of appropriateness and effectiveness needed to be solved. This paper addresses questions of its applicability to ordinary programming for Very Long Instruction Word machines. We developed practical methods of exploiting this parallelism (e.g. memory anti-aliasing). To justify and better understand the dynamic interaction between trace scheduling and anti-aliasing, we designed a more formal model in which we proved the correctness of trace scheduling and showed that it terminates. This in turn allowed us to analyze our flow information requirements. Finally we addressed the problem of ambiguous memory references which cannot be resolved at compile time.

1 Introduction

Trace scheduling[10] is a global compaction technique for transforming sequential programs into parallel code. When this investigation began, trace scheduling was unimplemented and many serious questions of appropriateness and effectiveness needed to be solved. We have attempted with this work to solve several questions of its correctness and (practical) applicability to ordinary programming for **V**ery **L**arge **I**nstruction **W**ord machines[11]. This report presents an overview of some of the issues involved in the process of building an optimizing compiler for VLIW architectures. An extensive discussion of these issues and more implementation details can be found in [18].

1.1 What Is a VLIW Machine?

VLIW architectures can be viewed as a type of extremely tightly coupled multiprocessors in which many statically scheduled, tightly coupled fine-grained operations execute in parallel within a single instruction stream. Such architectures have been discounted in the past mainly because of the extreme difficulty to generate good code even for existing machines of this kind such as the FPS164, in which the parallelism is quite modest.

1.2 Is There Enough Parallelism to Be Exploited?

To ascertain that the parallelism existing in ordinary sequential code is large enough to justify the building of Very Long Instruction Word machines, we have designed experiments which measure the maximum global parallelism potentially available in such programs, and found a large degree was available. Earlier work dealt only with local compaction[19] or hardware oriented schemes [12]). Because of this the parallelism reported was either too small (2-3 speedup factor over sequential execution) or far too expensive to be deemed exploitable by a realistic machine.

Yet other researchers[3],[4],[14] have measured the parallelism obtainable from ordinary programs by various compilation techniques. While these measurements produced better results (over an order of magnitude speedups) they were tightly related to specific methods and architectures. While encouraging, they were not directly relevant to our goals.

1.3 Exploiting parallelism using Trace Scheduling

Without a global compaction technique which could exploit the large degree of *potential* parallelism that our experiments have found, VLIW architectures exhibiting speedups factors of 10-30 would remain impractical.

Trace Scheduling, tries to overcome the limitations of basic block boundaries (particularly in scientific code). Thus it has the potential for efficiently exploiting relatively large VLIW machines, by performing massive numbers of code motions in order to fill instructions with operations that come from widely separated places in the program. These code motions are restricted by data precedence, which may be ambiguous in the case of array references.

As it turns out, disambiguating memory references accurately is crucial to the Trace Scheduling process, since a major part of the parallelism available in scientific code involves operations on arrays. Indeed, preliminary experiments have indicated that without an effective disambiguation mechanism the performance of a trace scheduling compiler will decrease dramatically.

1.4 Disambiguation and Trace Scheduling

From the above discussion it becomes clear that too conservative an approach to disambiguating array references will result in a major bottleneck, while too liberal an approach may yield incorrect results. To attempt to disambiguate as many of these references as possible, we have implemented an anti-aliasing system which relies on conventional and unconventional data-flow analysis methods, together with range analysis. This disambiguation module has to interact with the trace scheduler which dynamically alters the program in a complex way, and thus the type of analysis and responses has to reflect and react to these changes.

1.5 Correctness issues

In the process of developing the compiler it became evident that to properly understand the functioning of the trace scheduler, the memory anti-aliasing modules and the complex interactions which result, we need to formalize the functions of the trace scheduler and prove the correct functioning of our system in the context of the formal model. The importance of this analysis is underscored by the fact that several non-trivial conceptual problems in trace scheduling have been identified and solved by this process and several improvements became obvious. Furthermore, efficient and correct methods of flow analysis for trace scheduling have been designed. This was critical to the feasibility of a realistic trace scheduling compiler.

1.6 Handling dynamically determined references

Finally we have approached the problem of memory anti-aliasing of references which cannot be effectively disambiguated at compile time. This involves the idea of runtime disambiguation, in which a simple disambiguation mechanism is compiled into the parallel code produced by the compiler. References which are dependent on runtime information (e.g. an array index as a function of some input variable), and thus cannot be disambiguated at compile time, will be successfully handled by this mechanism.

2 Previous work

Before the advent of trace scheduling, VLIW architectures were discounted as too irregular and only suited for special purpose, limited parallelism applications. The belief that only a limited amount of parallelism exists in ordinary programs and that any such paralellism is mostly regular, has also played a role in the neglect of VLIW machines. Because of this, previous work done in the field of compilation of ordinary programs for execution on very parallel machines was centered around vector and pipeline processors. Important work in this domain was undertaken at the University of Illinois at Urbana, where D. Kuck and his group developed a system called Paraphrase, whose main goal is to generate code for fast, parallel machines.

The system uses a series of **source level** transformations in order to enhance the potential for finding parallelism:

- Induction variable substitution
- Renaming
- Expansion
- Forward substitution
- Loop distribution/fusion/interchanging.

As far as the parallelization proper is concerned, they use an extensive form of global data-dependence analysis called pi-partitioning and avoid the use of flow information. Once the program is partitioned into pi-blocks (maximal set of blocks cyclically connected by dependencies), the above mentioned techniques are applied in order to remove some of the dependencies, and then vectorizable code is identified and transformed to actual vector operations. A disambiguation mechanism, developed by Banerjee, is incorporated in their system and is used for the building of the data-dependency graph. Since the goal of the system is to find vectorizable operations, it tries to identify recurrences, break them into simpler ones and extract array operations from loops. It does so using a combination of software (such as the hyperplane method - similar to Lamport's method[16]) and hardware methods. In order to overcome the problem of conditional statements they use several other transformations and special purpose methods oriented towards special hardware (such as vector bit operations and hardwired recurrence solvers). In addition, pi-blocks can also be distributed to separate processors in a multiprocessor environment.

Our techniques differ from the above in several important aspects:

- We aim at a very fine level of granularity. This increases the potential for exploiting parallelism even in code which is not vectorizable, albeit at the cost of more analysis.

- Our approach works uniformly across conditionals and thus requires fewer special purpose transformations. This is a definite advantage, particularly in the case of less regular code.

- We use a simple data-dependency analysis coupled with sophisticated flow-analysis to achieve the accuracy required for a very discriminating anti-aliasing mechanism, which is crucial for fine granularity. This mechanism, coupled with loop unwinding achieves good performance and generality.

- Because of all of the above, and the changes introduced in the process of trace scheduling, there is a need for efficient dynamic interaction between the disambiguator, the trace scheduler and the code generator. Thus data-dependency and flow information has to be dynamically maintained.

2.1 Other Related Work

In the above we have outlined previous techniques for exploiting parallelism in ordinary sequential programs The disillusionment of researchers with exploiting parallelism in ordinary programs has led to several other approaches, which while not directly related to the work presented here, should also be mentioned.

A more radical approach, that of **dataflow models** developed primarily at MIT[7], UC Irvine[2], Utah[13],[6], and in France[5], essentially abandoned the hope of extracting high parallelism from ordinary programs using sophisticated compilers. Instead, they rely on highly parallel languages (functional or dataflow) combined with fine grained runtime parallelism and completely decentralized control. A related approach is that of reduction machines/languages[20].

Yet another method of parallelism exploitation has recently resulted from the new VLSI technology and the falling cost of hardware. This approach, called **systolic arrays**, was developed at CMU[15] and uses a large number of very tightly coupled processors to solve a given problem. The execution of the code proceeds in a highly synchronized manner. Systolic arrays which are custom tailored to a given problem may become attractive as the cost of VLSI falls and the performance increases.

3 Our Experiment

3.1 Design Issues

In the design of the experiment several decisions had to be made:

- **A language** had to be chosen, which will present us with realistic granularity for our target architectures (more than a high level language) but will also be more general than a particular assembly language which would have burdened the experiment with unnecessary detail and tie the results to a particular machine. For this purpose we choose an intermediate language called N-address code (NADDR), similar to the 3-address code used as compiler intermediate languages[1].

- The domain from which our set of samples were picked is that of scientific applications, which we felt is the most appropriate, for several reasons. First, such programs are very CPU intensive, and could benefit from any achievable speedup, particularly for real time applications. Furthermore, good compaction methods being expensive to run, they would be most useful in cases where programs have relatively large life spans and/or speed is critical. Another consideration in choosing our domain is the extent to which programs lend themselves to compaction under practical constraints. Scientific programs seem to offer the largest potential in this respect, since their control structures are relatively simple, and the conditional jumps are heavily biased towards one of the possible paths, making them ideal for trace scheduling. Finally, of significant importance for the validity of our results is the fact that the flow of control in our sample domain is not significantly influenced by the actual data on which the programs run.

- **An oracle** that would eliminate the inhibition of parallelism due to conditional jumps and ambiguous memory references had to be devised. The accurate prediction of conditional jump directions is crucial to overcome the restrictions on parallelism imposed by block boundaries, while the accurate disambiguation of memory references allows wide code motions which may otherwise be inhibited by an overconservative approach. (In scientific code these references account for a large part of the available parallelism, making precise anti-aliasing crucial).

3.2 Implementation Concerns

To simulate an oracle that will do the above we have designed two modules: an interpreter and a scheduler. The interpreter accepts the n-addr code generated from actual (Fortran or C) code by a translator program, and executes it instruction by instruction. The stream produced is passed to the scheduler which places each instruction at the earliest possible level for execution, based on the dependencies between the current instruction and the previously scheduled ones. At the time an instruction is ready to be scheduled, the code has executed all the way up through that instruction, all previous branches have been determined and all earlier indirect memory references have been resolved. The two pass nature of this process gives us our oracle. Since we were interested in experimenting with realistic (large) programs, building full dependency DAG's would have required too much space. Fortunately, this was not necessary. In our model, in which the only dependencies are data-dependencies, it suffices to keep track of the schedule level at which a variable was last read/written. Then when an instruction is considered for placement in the schedule, the variables read and written by the instruction are checked against the stored information, and the instruction is scheduled at the first level consistent with all the data-dependencies. Thus in effect we only keep a front line of the DAG, which is dynamically updated in the process of building the schedule. (Essentially the same approach is used in the actual BULLDOG compiler). Another advantage of this approach is that instructions can be scheduled on the fly, without keeping the whole schedule in memory. The schedule obtained in this way is guaranteed to be optimal, given the assumption of infinite hardware resources.

The ability of the system to cope with large, real programs has been further enhanced by the practical implementation. Each module has been built as a separate process, which runs in its own private environment (and thus its own private address space). The interprocess communication takes place through pipes, implemented through connections on a local area network.

3.3 How much potential parallelism *really* exists?

In the experiment described above which effectively disregards basic block boundaries and reference ambiguity, we have found huge amounts of parallelism available. Our measurements were made on standard FORTRAN programs in common use. The actual programs tested averaged about a factor of 90 parallelism. It ranged from about a factor of 4 to virtually unlimited amounts, restricted only by the size of the data.

3.4 Can This parallelism be exploited?

An important question is how much of this parallelism can actually be found and used by a real code generator. In the experiments, an oracle is used to resolve dynamic questions at compile time. It tells us which way jumps went and whether indirect references are to the same or different locations. Our compiler attempts to get the effect of the oracle at compile time with static index analysis and dynamic estimates of jump probabilities. We argue that most scientific code is so static that the oracle is fairly realistic. A real trace-scheduling code generator might very well be able to find and use much of this parallelism. The infinite hardware assumed for the purpose of the experiment cannot of course be provided in a real machine. However our results provide an upper bound on the *exploitable* parallelism, and thus an indication of the required width for an effective VLIW machine. We can realistically expect such a machine, in conjunction with a Trace Scheduling compiler to exploit a sizable fraction of this parallelism even in cases where the potential parallelism is greater than the width of the actual machine (TS will provide a graceful degradation in such cases). Indeed, the preliminary results obtained in the development of the ELI-512 and its Bulldog compiler substantiate this claim[8], [18],[21],[9].

4 Memory Anti-aliasing - Disambiguation

4.1 Why do we need disambiguation?

Trace Scheduling replaces block-by-block compaction of code with the compaction of long streams of code, possibly thousands of instructions long, by treating them as basic blocks and then inserting remedial code where this would causes problems. To take advantage of the parallelism made available in the process, it is necessary to do massive numbers of code motions and fill instructions with operations that come from widely separated places in the program. These code motions are restricted by data precedence, which has to be preserved to insure correctness.

4.2 The Problem of Memory Disambiguation

For example, suppose our program has the steps:

1. Z:= A + X

2. A:= Y + W

Our code motions must not cause (2) to be executed earlier than (1). So the scheduler must know the Data-Dependence Graph of the program for the scheduling process to yield correct results. Ambiguity arises however in the case where A is an array reference:

1. Z := A[expr1] + X

2. A[expr2] := Y + W

Whether (2) may be done earlier than (1) is unclear. If exprl can be guaranteed to be different from expr2, then the code motion is legal otherwise it is not. This will be further complicated if one or both of the references are functions of loop indexes. Answering the question of whether 2 references are independent of each other is the problem of Memory Anti-Aliasing. With other forms of indirections (e.g. pointers) there is little chance of success at compile time, unless they are severely constrained (turned in effect into array references).

4.3 Where and How Well Can Anti-Aliasing Succeed?

Indirect references in inner loops of scientific code are mostly array references, and such code (in conjunction with various techniques such as loop unrolling) usually offers the greatest potential for parallelism. Thus the very accurate disambiguation of such references is crucial to the success of our compiler and VLIW machines, since too conservative an approach will lead to inefficient use of the machine (and small speedups). Our systems achieves as fine a discrimination as possible at compile time by using such conventional flow-analysis techniques as reaching definitions and non-conventional ones such as variable-folding and range analysis to refine the solution to the Diophantine equation obtained from comparing the given references.

4.4 The Memory Anti-Aliasing Mechanism and its Implementation

We have implemented a system which given two array references:

1. ... A[exprl] ...

2. ... A[expr2] ...

attempts to solve the Diophantine equation obtained by equating the indexing expressions. To achieve maximal accuracy in the disambiguation process, extensive flow-analysis is used to refine the results of these comparisons, as well as other methods, which attempt to reduce the number of variables involved in the equations. The system consists of the following main modules:

- **Flow graph builder** - constructs the program flow graph, using conventional techniques. This graph is used for the flow analysis that follows.
- **Modified live-dead analyser** - needed for conditional dependencies. The modifications are efficiency oriented, such as storing rather than recomputing local information. Special handling of array references to try to improve accuracy is also attempted.
- **Modified reach analyser** - used by the following modules. The modifications are mainly efficiency oriented, such as storing rather than recomputing local information and preserving only information needed for our purpose. Special handling of array references to improve accuracy is also attempted.
- **Conventional constant folder** - The constant folding is performed on the actual operations and is thus incorporated in the generated parallel code.
- **Generalised constant folder** - folds constants across conditional jumps and rejoins. Similar to the variable folding below. This is

done to increase the accuracy of the predictions of the disambiguator, and is only performed on the reaching definitions, and not the actual code, since the expressions generated are not executable. For efficiency, this processing is performed only where and when required and the information is only stored for as long as it is needed.

- **Variable folder** - symbolic variable manipulation, similar to constant folding, across conditional jumps and rejoins. Reduces all expressions of interest to combinations of irreducible variables and constants. For example:

```
                                   Reaching definitions at (*)
       a=b                         a  ....      b | d+2
       c=d+1                       b  ....        ?
       if (x.ne.0)                 c  ....       d+1
          then a=c+1               d  ....        ?
       e=a+b+c                     e  ....2b+d+1 | 2d+b+1
   (*)                             x  ....        ?
```

For efficiency, this processing is performed only where and when required and the information is only stored for as long as it is needed.

- **Loop analyser** - identifies loops, self-referencing variables and recurrences. It then standardizes as functions of unique induction variables (one per loop), up to two mutually referencing variables. Handling nonlinear recurrences and multiple mutual references in general is computationally expensive, and in our experience (given the other types of analysis which we perform) is hardly ever needed in practice.

Example:

```
    Defs reaching the loop: a=A0, b=B0, c=C0.
     ----                   Reaching defs after loop analysis:
loop| a=b+6                 a... B0+6+(i-1)(3+6)
body| b=a+3                 b... B0+i(3+6)
    | c=b+1                 c... [B0+i(3+6)]+1
     ----                   Where i is a "normalized" index:
                                   i=1,#iterations
```

- **Algebraic expression normalizer** - simplifies symbolic expressions and puts them in a canonical form. This enables the actual comparison mechanism to do a good job.

- **Simple range analyzer** - finds loop bounds and array sizes, notices odd/even or incrementally decreasing/increasing ranges and expressions. This information is then passed to the loop analyzer and to the comparison mechanism. It serves to refine the results of the disambiguation, and to improve accuracy.

- **Memory reference comparator** - Uses one of several methods to disambiguate two given references, depending on the information obtained using the above modules:

  1. Both references are constants or constant expressions - evaluate and compare.

  2. Multiple constant values for each reference - find intersection of the two sets.

3. Same variable in both references - solve one equation in one unknown.

4. More than one variable - solve Diophantine equation(s).

Note that cases 1-3 yield precise answers (i.e. total disambiguation) whereas case 4 may not, by itself remove the ambiguity. To refine the accuracy of the results, we discriminate between the following cases:

- Both references are in loop free code.
- One or both references are in loops, but we are interested in particular iterations.
- One or both references are in loops and we are interested in whether collisions are at all possible.

Several of these modules encounter special problems which arise from the dynamic changes made by the Trace Scheduler to the original program. These changes may impair the accuracy, or even the correctness of the anti-aliasing system. We will address these concerns after presenting more rigorously the functioning of the trace scheduler and showing its correctness. Then we will be better able to understand what influence the dynamic changes introduced by the trace scheduler may have on the analysis of the code, where the functioning of the anti-aliasing mechanism may be impaired and the methods which should be used to ensure correctness and accuracy.

5 Correctness of Trace Scheduling

Correctness issues kept reoccurring throughout the implementation of the original Bulldog compiler. These problems concerned the legality of the transformations undertaken by Trace Scheduling on one hand and their possibly disastrous effect on the legality of the flow information obtained statically on the other. Because of the shaky, intuitive nature of our arguments, discussions trying to resolve these problems led nowhere, and whatever issue seemed to have been solved in one such session would be brought back into question at the next occurrence of a bug. After some time, it became evident that the only way out would be to define a more rigorous model for Trace Scheduling, prove the correctness of the transformations and settle the arguments in this formal context. Since we were dealing with an existing system, our model and proofs had to stay very close to the implementation in order to provide realistic (and useful) information. The lengthy and detailed description of the model and proofs can be found in [18]. In this section we will only provide an outline of the formalization and proof of correctness proper.

5.1 Informal Description of Trace Scheduling

Very informally, Trace Scheduling replaces the block-by-block compaction of code with the compaction of long streams of code, possibly thousands of instructions long. This is achieved using the following approach:

1. **Traces**

   Assume we have loop-free code, with no back edges (Loops are handled naturally by not having traces picked across backedges). Jump predictions are used to select streams with the highest probability of execution, which are then compacted as if they each constituted one big basic block.

2. **Preprocessing**

   Preprocessing is done to prevent the scheduler from making absolutely illegal between block code motions, i.e. those that would clobber variables which are live off the trace. The constraints on code motion are encoded into the data precedence graph via edges from the conditional jump off the trace to any later operation that could clobber the variable.

3. **Postprocessing** After scheduling has been completed on a stream, the scheduler has made many code motions which, while potentially legal, will not correctly preserve jumps from (or rejoins to) the stream to the outside world. To make these code motions actually legal, a postprocessor inserts new code at the stream exits and entrances. Without this ability, available parallelism would be unduly constrained by the need to preserve jump boundaries.

The most frequently executed code beyond the stream, including the new state recovery code, is then compacted as well, possibly producing more state recovery code.

5.2 A More Precise Description of Trace Scheduling

Given a program graph P, the trace scheduler converts it to another program graph, P' by selecting a trace, *t* in P and replacing it with a new one, t', obtained as follows[1]:

Trace Picker

Chooses (by any method) a trace $t = (n_1, ..., n_m)$, where the n_i's are uncompacted nodes (operations) in P[2].

Compactor

1. **Trace Compactor (TC):**

   Given t, produces a new (compacted) trace, $t' = (N_1, ... N_{m_1})$ where:

   $N_1 \equiv$ all operations without dependency predecessors in t.

   $N_k \equiv$ an operation is placed in N_k if it has either a non-strict predecessor in N_k or a strict predecessor in N_{k-1}, and has no dependency predecessors in subsequent N's[3]. All the N_i's are marked "compacted". (The edges in the new trace t', are defined by CR below.)

2. **Split Compensation (SC):**

   For every branch (CJ_i, S_i), adds a sequence of nodes,SC_i, and edges to the new graph, between $N_k(CJ_i \equiv n_i \in N_k) and S_i$:

   $SC_i \equiv$ copies of all operations preceding n_i in t, which don't precede it in t'. (The edges between these nodes are defined by CR below.)

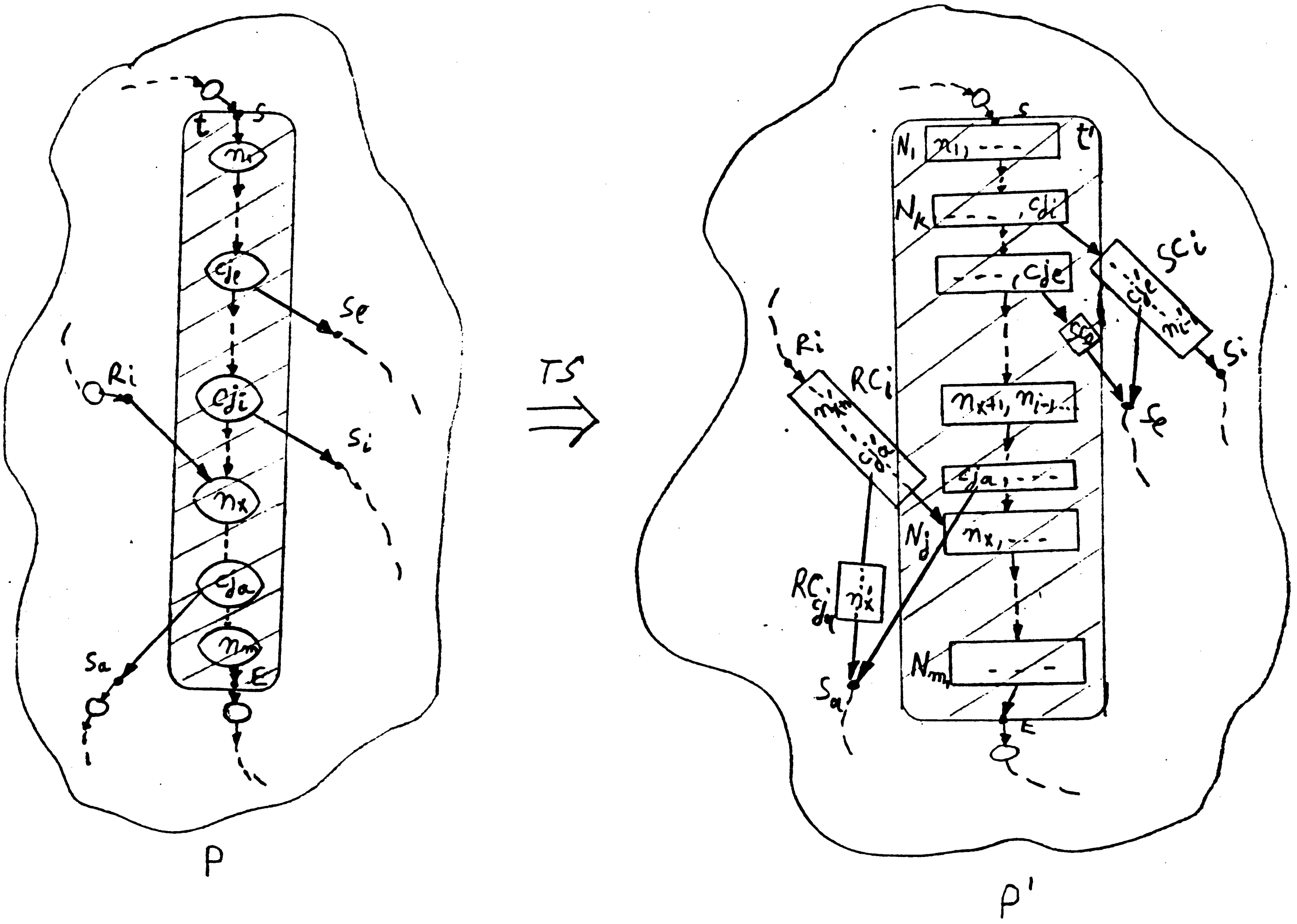
3. **Rejoin Compensation (RC):**[4]

[1] For examples of the terms used in this discussion (e.g. trace, program graph, cj, branch, rejoin/split points, entry/exit points) see figure 1. The precise definitions can be found in [18].

[2] In t each node n_i contains exactly one operation; in t' each N_i can contain more than one operation.

[3] A node $n_{i+\epsilon}$ is strictly dependent on n_i, if $n_{i+\epsilon}$ has to execute after n_i to produce the desired output; $n_{i+\epsilon}$ is non-strictly dependent on n_i if $n_{i+\epsilon}$ can execute together with, or after n_i, to produce the desired output.

[4] The rejoin compensation described in previous papers dealing with trace scheduling was



Before the compaction. **After the compaction.**

- *P and P' are program graphs.*
- *S,R_1 are entry points, S_l, S_i, S_a, E are exit points.*
- *cj's are conditional jumps, (cj_i, S_i) is an (off-trace) branch.*
- *Trace t consists of all nodes (and intervening edges) from S to E.*
- *A trace-path p through t is $S \rightarrow S_l$. The equivalent trace-paths of p in P' are $e_1 \equiv S \rightarrow cj_i \rightarrow cj_l \rightarrow S_l$ and $e_2 \equiv S \rightarrow cj_i \rightarrow cj_l' \rightarrow S_l$.*

Figure 1: Sample trace scheduling (one trace)

RC1

(a) Given a rejoin to $n_x \in t$, ($n_x \in N_k \in t'$), the *new rejoin* in t' is to N_j, where $j \equiv$ highest point in t' below which only operations originally at or after n_x are found, and no cj's from below n_x are below j.

The rejoin compensation adds a sequence of nodes, RC_i, and edges between R_i and the new rejoin to N_j in the new graph:

(b) $RC_i \equiv$ copies of all operations which used to appear below the old rejoin, but are not below the new one in t'. (The edges between these nodes are defined by CR below.)

RC2

(a) Given a rejoin to $n_x \in t$, ($n_x \in N_k \in t'$), the *new rejoin* in t' is to N_j, which is the highest point in t' below which only operations originally at or after n_x are found.

The rejoin compensation adds a sequence of nodes, RC_i, and edges between R_i and the new rejoin to N_j in the new graph:

(b) $RC_i \equiv$ copies of all operations which used to appear below the old rejoin, but are not below the new one in t'. (The edges between these nodes are defined by CR below)

Furthermore, for all $cj_a \in RC_i$, an additional sequence of nodes and edges is inserted in P' between cj_a and its target S_a. Assuming cj_a

found to be incorrect. The alternative transformations presented in this section are correct.

corresponds to n_l in t, and to r_y in RC_i, then:

(c) $RC_{cj_a} \equiv$ copies of operations originally above n_l which aren't in RC_i.

4. **Continuation Resetting (CR):**

   For each new node N_i, we define new edges to the next following (new) node, or if no such node exists, the corresponding split/rejoin/exit point. The continuation resetting affects only the on- trace/trace-path branch of Cj's; the off- trace/trace-path branch is unchanged.

5.3 Correctness of Trace Scheduling

To prove the correctness of trace scheduling we have to show:

- **Partial Correctness**
- **Termination**

Definition: TS is **partially correct** if for every program graph P and input i, $output(P(i)) \equiv output(TS^{(n)}[P(i)])$.

Definition: Trace scheduling is said to **terminate** if for any input program graph P, a finite number of applications of TS to P, always result in a program graph containing no uncompacted nodes.

5.3.1 Partial Correctness

We prove that trace scheduling is correct by showing that compacting one trace t in P, yields a new program P' which is indistinguishable from the original one in terms of its input/output behavior. Then we generalize by induction to any (finite) number of trace compactions. To establish the correctness of a single application of trace scheduling, we first define some state-preserving properties which, if satisfied throughout the compaction process by each and every *trace-path* p in P and its *equivalent* trace-paths $\{e_1, .., e_n\}$ in P', will guarantee the correctness of the transformation[5].

The properties which are sufficient to ensure state preservation are:

- **(a) Dependency Correctness:**

  1. For each node in p there is one (and only one) semantically equivalent operation in some node of each e_i.
  2. For any pair of such nodes (a, b), if b is dependent on a in p, and $a \in N_k, b \in N_j$, then $k < j(if strictly\ dependent)$ or $k \leq j$ $(if non - strictly\ dependent)$ in each e_i ($\equiv$ **dependency precedence**). Note that the above also holds for conditional jumps.
  3. Operation o, which appears in a node of e_i but not of p must:
     - Not define variables live on exit from e_i.
     - If o depends on $a \in p$, then (a, o) must satisfy dependency precedence in e_i.

- **(b) Control Correctness:**

  1. If a node Cj_i exists in p, with a branch (Cj_i, S_i), then a branch (N_k, S_i), (where N_k contains Cj_i), must exist in e_i, with the same direction as the one in p (i.e. identical test resolutions must lead to equivalent paths being taken in P and P').

[5]Intuitively, a trace-path is a path from an entrance point into the trace, to an exit point from the trace (see also figure 1).

We then proceed by a case by case analysis, to demonstrate that any legal application of trace scheduling transformations will indeed preserve these properties[6].

We then argue that since P' preserves the semantics of the original P, applying TS to P' also preserve correctness, and thus, $output(P^{(n)}(i)) \equiv output(P(i))$ for all i (assuming TS halts after n steps).

5.3.2 Termination

To prove that the process of trace scheduling itself terminates we first restrict ourselves to DAG's (directed acyclic graphs). We show that any legal *composition* of trace scheduling transformations will restrict the ability of the trace scheduler to pick new traces[7], and thus force termination.

More precisely, given a DAG G of height max, we form an n-tuple, $< l_{max}, ..., l_1 >$, where l_i is the number of paths of length i in G. The elements of the tuple are sorted (from left to right) in the order of decreasing lengths. That is, l_k precedes l_i *iff* $k > i$. Furthermore, a lexicographic ordering ($\prec$) is defined on the tuples.

All the transformations which may occur in the course of Trace Scheduling are shown to only transform our n-tuple (in the worst case) in the following way:

$$< l_{max}, ..., l_k, ..., l_1 > \Rightarrow < l_{max}, ..., l_k^-, l_{k-1}^+, ..., l_1^+ >$$

[6]Note that in the process of compacting a trace, new trace-paths may be created which also have to satisfy the state-preserving conditions with respect to their corresponding original paths.

[7]Looking at individual transformations is not sufficient since some of them appear to inhibit termination when examined in isolation.

That is, each transformation will reduce the number of paths of length k by at least 1, while possibly increasing the number of paths of length less than k by some finite number, where k is the length of the longest path affected by the transformation.

Since $< l_{max}, ..., l_k, ..., l_1 > \succ < l_{max}, ..., l_k^-, l_{k-1}^+, ..., l_1^+ >$, proving the above is sufficient to show the process will have to terminate[8].

Finally, trace-fences are all we need to enable the termination argument for DAG'S to apply to arbitrary graphs: Since traces can't be picked across trace-fences, we can treat trace-fences as start/exit points for paths, and proceed as above.

5.4 Windfall Profits Resulting form the Above

As a result of the better understanding of the Trace Scheduler gained in the process of developing the above model, a significant deficiency in the compensation mechanisms has been identified and eliminated. Several other conceptual problems have also been exposed and solved by this process. Furthermore several generalizations/improvements have suggested themselves as a result of the insight gained:

- Only copy as compensation operations which define variables which are used (live) below split-points (for splits) or rejoin points (for rejoins). This would require semi-dynamic live-dead analysis (which is needed in any case). This will also require an extra split-compensation step, similar to (c) in rejoin-compensation, but would have the aesthetic effect of making split/rejoin compensation symmetrical.

[8] This proof method is similar to Manna's [17].

- Conventional Live-dead analysis is not satisfactory for our needs in establishing conditional jumps dependencies with respect to array references. Static live-dead analysis is simply incorrect, while fully dynamic live-dead analysis is too expensive (and unnecessary).

- Conventional handling of array references in flow analysis methods is more conservative than necessary. In the following section, better methods are described.

- Rather than using conservative compensations, which only allow safe movements of code, we could use "undo" compensations, which will not restrict parallelism. This is trickier, since it requires reversing the effects of already executed operations.

6 Flow-analysis Issues

For the process of Trace Scheduling to preserve correctness certain types of flow information (e.g. live-dead) is required. Furthermore, in order to achieve optimal results in the compaction process the disambiguation mechanism is crucial and it in turn depends for its accuracy on reach analysis. Since flow analysis can be computationally expensive, we would like to be able to use static flow information derived from the original program, or at least semi-static information (i.e. incrementally updated between TS applications). However in the presence of the extensive global code motions performed by the Trace Scheduling algorithm, it is not immediately clear whether static flow analysis can be used directly or in conjunction with periodic updates, while preserving the correctness of the transformed program. Indeed, it may intuitively appear that the original program is changed to such an extent, that anything - short of full dynamic analysis after and during each trace compaction - is totally useless.

6.1 This feeling is quite wrong

We have shown that this intuitive feeling is wrong, and that static or semi-static flow information can safely be used. We have also shown the cases were incremental updating is needed and how it can be used without loss of accuracy with respect to the far more expensive dynamic analysis. Finally we describe a live-dead analysis algorithm which enhances the discrimination between array references in live-dead information and hence significantly improves the performance of the Trace Scheduler.

6.2 Our Results

- **Static reach analysis is enough** for anti-aliasing. (This follows easily from the correctness of Trace Scheduling). Static reach may be somewhat conservative if no updating is used, as can be seen in the following example:

```
Before compaction                After compaction


  a=...   a=...                  a=...            a=...
     \   /                         |                |
      v v                          v                v
(*)  ...=a...             (*)   ...=a...    (*')  ...=a...
        |                          |                |
        v                          v                v
2 defs reach (*).             Since we now have two separate
                              paths, (*) and (*') are reached
                              by only one definition each.
```

- **Live-Dead information at the Entrance/Exit points of a trace doesn't change** during the trace compaction since the only modifications to the program-graph appear in between entrance and exit points and all operations on any trace-path being preserved (in dependency order), the relative order of definitions/uses of the same variables on any given trace-path is also preserved.

- **Totally static live-dead information is not enough** for trace scheduling, as can be seen in figures 2,3.

```
o R           o S
 \            |
  \          i:=1
  x:=1        |
    i:=x      |
      \       |
       \      |
        \   a:=i+1
         '> a:=a+1
            b:=a
              |
            cj(i)
              |  \
              |    \
            b:=b+1   \        a is dead
              |        '->o b is live
              v           S1
              o E
            a is dead
            b is live
```

Figure 2: Incorrect Trace Compaction as a Result of Erroneous Live-dead.

(a) Before compaction of trace $S \rightarrow E$.

```
 o R                 o S
  \                    |
  x:=1              i:=1
     i:=x              |
       \               |
      cj(i)            |
     /     \           |
(*)a:=a+1   \       a:=i+1;cj(i)
   b:=a      `---->a:=a+1      \
    \              b:=a       a:=a+1
     \                 |        b:=a
      `--------------|-----.      \
                       |      \      v
                       |       `--> o S1
                   b:=b+1          a is dead
                       |           b is live
                       v
                     o E
                  a is dead
                  b is live
```

Figure 3: Incorrect Trace Compaction as a Result of Erroneous Live-dead.

(b) Before compaction of trace $R \rightarrow (*) \rightarrow S1$

The original trace being compacted is $S \rightarrow E$ (figure 2). After the compaction, we obtain the graph in figure 3. If in this new graph we choose to compact the trace-path $R \rightarrow S1$, the static (original) live-dead information (on the branches of the cj) will not prevent operation (*) from moving across the cj, which is incorrect.

- **No update of live-dead information is needed** during the trace compaction since operations in split compensation that may change live-dead information appear below the cj causing the split in the compacted trace body, and thus block any operation which would potentially violate conditional dependency[9] (as a result of not having dynamically updated live-dead). Thus such operations are not even allowed to reach the cj.

- **Incremental (after trace compaction) updating of live-dead information is both correct and accurate** for the purposes of trace scheduling, since any operation killing a variable which was originally live on the off-trace branch of the cj, as a result of being part of split compensation, will appear below the cj in the compacted trace body, and thus will block all operations which may have otherwise moved above the cj (if the only thing stopping them were the conditional dependency at the cj). Thus the obvious updating (i.e. modifying the static live-dead information by propagating the changes upward from exit points) will be accurate. (Its correctness is ensured by the correctness of TS which implies that no changes in live-dead information can

[9] An operation *o* is conditionally dependent on a conditional *cj* if *o* writes a variable which is live on *cj*'s off-trace branch.

occur at entrance and exit points of a trace as a result of compaction).

- **Ordinary live-dead analysis in the presence of array references is not really appropriate** for the purposes of trace scheduling, as can be seen in the following example:

```
                    |
                    v
              ---------------
         (1)| if condition|
              ---------------
              /             \ <- p1
             V               V
        ------------   ------------
   (2)|A[x+1]:= 8|  (4)|A[2*x]:=10|
        ------------   ------------
              \             /
               v           v
               ------------
          (3)| PRINT (A)|
               ------------
```

The question is, can statement 2 in the trace 1,2,3 move above statement 1?

Ordinary live-dead analysis[1] does not deal with arrays explicitly. The most natural (and naive) way to deal with arrays is to simply try to propagate array names as variables. However one must be careful in defining *conservatively* what will be a definition and a use, as the meaning of "conservative" may vary with the application; for trace scheduling "conservative" would mean allowing arrays to be live whenever in doubt. In the above example, the conservative way out is to have all of array A being live at point p_1. The meaning of "conservative" in our case is a function of the way the disambiguator, and through it the trace scheduler, makes use of live-dead information: operations defining variables which are live on the off-trace branch of statement (1) in our example, should not be allowed to move above (1) on trace. It is therefore preferable in computing live-dead information for this purpose to err by allowing all of A to be live in cases were we are unsure of which parts of it are still live at p_1, rather that claim that all of A is dead at p_1. The former is merely conservative, while the latter will allow incorrect compactions to occur.

While correct, the above approach may be too restrictive, since it would drastically hinder the movement of operations involving arrays[10].

Several simple solutions to this problem come to mind (e.g. use range information, propagate each element of the array as a single variable, etc). These solutions are either too expensive, not accurate enough or even incorrect.

What we *really* need if we are to do this right, is a mechanism which would allow *symbolic* manipulation of arrays, with flexible degrees of granularity varying from full array down to single elements. This mechanism could use the following types:

[10]This is less bad than it sounds since the conditionals which would restrict such movement will also be able to move upwards, improving the ability to achieve good compaction.

- Subrange
- Odd
- Even
- All (every)
- All-But

and logical operators such as OR, AND, NOT, and MINUS.

Once we integrate this mechanism into the live-dead analyzer, we can apply the full power of the disambiguator to determine whether or not a code movement past a conditional jump is legal. For example, operation 2 above could not be allowed to move above operation 1.

7 Runtime Memory Anti-aliasing

The idea of runtime disambiguation (RTD) deals with memory anti-aliasing of references which cannot be effectively disambiguated at compile time. To achieve this, part of the disambiguation mechanism is integrated into the parallel code produced by the compiler. In this way, even references which are dependent on runtime information and thus cannot be disambiguated at compile time (e.g. an array index as a function of some input variable), will be handled by this mechanism.

In order for this mechanism to be effective the code introduced has to be simple and highly compactible. The solution we propose is as follows: Given two references :

```
(a)  A[i]
      .......

(b)  A[j]
```

on a path, and a possible conflict between them (i.e. i possibly equals j), we transform the above code segment to:

```
(a)         A[i]
            ..........
            if-ine i j  .9  l1 l2
        l1: assert i#j
        l2:............
(b)         A[j]
```

This will force the trace (a,l1,b) to be given priority in the compaction process and the assertion at l1, will be used as a compiler directive, allowing a,b to be scheduled independently of each other. Of course, the off-trace branch will take care *at runtime* of the case in which i=j. The **if statement** depends only on the computation of i and j may potentially be placed early in the schedule, (quite possibly for free – if resources are available). Thus if the conflict between the references is either nonexistent (but the disambiguator cannot establish that at compile time) or occurs rarely (e.g. i,j are used to traverse A in opposite directions), the speedup resulting from the use of runtime disambiguation can be significant.

7.1 Practical Considerations/Improvements

While the above describes the essential idea of runtime memory anti-aliasing, the actual implementation has to take into account several other factors:

- Renaming is necessary both for correctness and efficiency.
- Correctness issues have to be addressed, since RTD alters the compaction based on assumptions which can only be checked at run time. Thus we have to ensure that the (compile time) code generated will be legal even if the assumption fails. Two possibilities come to mind:
  1. Never allow operation (b) to move at or ahead of the test, and readjust the new rejoin point of the test ultra-conservatively to be after both (a) and (b) in the trace. This solution is straightforward but may severely limit the code movement and thus the benefits from RTD.

2. A more general solution in which compensation is introduced to restore the correctness of the compacted code. This scheme does not restrict code motion unnecessarily and results in a natural extension of trace scheduling.

Both approaches have been implemented and as expected, the second performs better than the first.

- Space considerations will require us to limit the code proliferation resulting from the insertion of conditional statements by RTD. Several methods can achieve this without undully constraining parallelism, such as using CSE and making the RTD mechanism smarter in deciding when and where code should be inserted (carried to the extreme this would mean 2 full passes of TS). We have implemented several such methods and optained good preliminary results.

In our experience to date with the system we have implemented for RTD, significant speedups can be achieved, ranging up to twice those obtained using only trace scheduling.

8 Conclusions and Directions for Further Research

Based on the work described, and our experience in the ELI project as a whole, we believe that the following claims have been substantiated:

- Sufficient parallelism exists in ordinary scientific code to justify the building of VLIW machines.
- Most of this parallelism is exploitable by the use of Trace Scheduling in conjunction with powerful memory anti-aliasing tools.
- Trace scheduling is correct and terminates, and its efficiency could possibly be improved at the expense of more complexity and space/time tradeoffs.
- RTD can improve the compactor's ability to deal with code which has unpredictable control flow, and thus enable it to handle more general types of code.

We also believe that the research described here and in [18] and the problems we had to face in the course of this project have opened several paths for further research:

- Applications of Trace Scheduling techniques to other parallel architectures.
- Runtime Disambiguation refinements/extensions.
- Other forms of Runtime Disambiguation.
- Generalizations of Trace Scheduling.
- Hybrid machines.

Bibliography

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

[2] Arvind and Kathail,V., "A multiple processor data flow machine that supports generalized procedures", Proc. 8th Annual Sym. Comp. Arch., ACM, May 1981, pp. 291-302.

[3] Banerjee U., S. C. Chen, D. J. Kuck and R. A. Towle, "Time and Parallel Processor Bounds for Fortran-Like Loops", IEEETC, September 1979 Number 9, pp. 660-670.

[4] Chen S.C. and D. J. Kuck, "Time and Parallel Processor Bounds for Linear Recurrence Systems", IEEETC, July 1975 Number 7, pp. 701-717.

[5] Comte, D., Hifdi, N., and Syre, J.C., "The data driven LAU multiprocessor system: results and perspectives", Proc. IFIP Congress '80, October 1980, pp. 175-180.

[6] Davis, A.L., "The architecture and system method of DDM-1: A recursively-structured data driven machine", Proc. Fifth Annual Symposium on Computer Architecture, 1978.

[7] Dennis, J.B., and Misunas, D.P., "A preliminary architecture for a basic data-flow processor", Proc. of the 2nd Annual Symposium on Computer Architecture, ACM, IEEE, December 1974, pp. 126-132.

[8] Ellis, J.R., "Bulldog: A Compiler for VLIW Architectures", Yale University, July 1984, Expected.

[9] Fisher J.A., Ellis J.R., Ruttenberg J.C. and Nicolau A., "Parallel Processing: A Smart Compiler and a Dumb Machine", The ACM Symposium on Compiler Construction, June 1984.

[10] Fisher, J.A., "Trace scheduling: A technique for global microcode compaction", IEEE TC July 1981 Number 7, pp. 478-490.

[11] Fisher J.A., "Very long instruction word architectures and the ELI-512", Yale University Department of Computer Science, Technical report # 253, December 1982, Also appeared in 10th Annual International Architecture Conference, Stockholm, JUNE '83.

[12] C. C. Foster and E. M. Riseman, "Percolation of code to enhance parallel dispatching and execution", IEEETC, December 1972, Number 12, pp. 1411-1415.

[13] Keller, R.M., Lindstrom, G., and Patil, S., "A loosely-coupled applicative multi-processing system", AFIPS, June 1979, pp. 613-622.

[14] Kuck,D.J.; Muraoka,Y.; and Chen, S.-C., "On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup", IEEETC December 1972, NUmber 12, pp. 1293-1310.

[15] Kung, H.T., "Why Systolic Architectures?", IEEE Computer, January 1982, Number 1, pp. 37-46.

[16] L. Lamport, "The Parallel Execution of DO Loops", CACM, Number 2, February 1974, pp. 83-94.

[17] Z. Manna, S. Ness and J. Vuillemin, "Inductive Methods for Proving Properties of Programs", CACM August 1973, Number 8, pp. 491-502.

[18] Nicolau, A. "Parallelism, Memory Anti-Aliassing and Correctness for Trace Scheduling Compilers", Yale University June 1984.

[19] G. S. Tjaden and M. J. Flynn, "Detection and parallel execution of independent instructions", IEEETC October 1970, Number 10, pp. 889-895.

[20] Treleaven, P.C., Brownbridge, D.R., Hopkins, R.P., "Data-driven and demand-driven computer architectures", ACM Computing Surveys, March 1982, Number 1, pp. 93-143.

[21] Ruttenberg, J.C., "Delayed Binding Code Generation for a VLIW Supercomputrer", Yale University, July 1984, Expected.