Parallelism, Memory Anti-Aliasing and Correctness for Trace-Scheduling Compilers

by
Alexandru Nicholau

YALE/DCS/RR—375
March, 1985

YALE UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

Parallelism, Memory Anti-Aliasing and Correctness for Trace Scheduling Compilers

Alexandru Nicolau

Yale University, 1984

Trace scheduling [12] is a 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. Chief among them was disambiguation, the act of determining at compile time whether two indirect references are to certainly different locations. This thesis demonstrates that disambiguation is practical, correct and can be done efficiently in the presence of the extensive code motions introduced by trace scheduling.

To demonstrate the practicality of disambiguation, a major implementation was undertaken which was part of the BULLDOG compiler for over a year. The effectiveness and necessity of the disambiguator became overwhelmingly obvious as the trace scheduling compiler was built. Turning it off made the parallelism we found decrease sharply.

A trace scheduling compiler does many code motions that dramatically change the flow of control as code is generated. Disambiguation, and indeed trace scheduling, can't work correctly unless flow analysis information is constantly updated. However a global dynamic flow analysis is far too expensive. Unfortunately there are no intuitive reasons for believing that that is not required. It was therefore necessary to formally analyze our requirements. Through this process we were able to show that incremental and local flow analysis is always correct for our purposes. In the process, a proof of the correctness of trace scheduling evolved as well.

Finally we studied new directions for disambiguation. In particular, a new technique, called Run Time Disambiguation was suggested and our implementation of this technique is described.

Acknowledgements

First and foremost I would like to thank my thesis advisor Professor Joseph A. Fisher for his encouragement and advice. His support and guidance have been of the utmost importance in the successful completion of this work.

I am also indebted to Professor Alan J. Perlis and Martin H. Shultz who were part of my readers' committee. Professor Perlis' insightful comments greatly helped in shaping this work into a better thesis.

Finally I would like to thank my friends and colleagues Neta Amit, John Ellis, Gregory Sullivan, Lenny Pitt, Doug Baldwin and Jerry Leichter for their many helpful comments and their interest in my work.

Of course, special thanks are due to my parents, who made this feasible in the first place.

Table of Contents

Chapter 1: Introduction 1
1.1 Overview of This Thesis 1
1.2 Very Long Instruction Word Architectures 2
1.2.0.1 Why Haven't These Machines Been Built for Massive Parallelism? 3
1.3 Trace Scheduling 4
1.4 The Problem of Memory Anti-Aliasing 6
1.4.1 Where and How Well Can Disambiguation Succeed? 7
1.5 Where Does This Work Fit in the Greater Scheme of Things? 8
1.5.1 Overview of the BULLDOG Compiler 8
1.5.2 The Role of this Thesis in the ELI Project 10

Chapter 2: PREVIOUS WORK 12
2.1 Introduction 12
2.2 What Were the Earlier Experiments? 13
2.3 Other Previous Experiments 14
2.4 Our Experiment 14
2.4.1 So What Was Wrong With the Earlier Experiments? 14
2.4.2 Trace Scheduling Conditional Jumps and Ambiguous References. . . 15
2.4.3 Using an Oracle to Measure Available Parallelism 16
2.4.4 Results 16
2.4.4.1 Global Speed-up Ratio 16
2.4.4.2 Basic Block Compaction 17
2.5 Compiling ordinary code for parallel execution: Parafrase 17
2.5.1 Construction of the Dependence Graph and Disambiguation. 18
2.6 Differences in our approach. 20
2.7 Other Related Work 21

Chapter 3: Memory Disambiguation 22
3.1 Background. 23
3.1.1 Why do we need disambiguation? 23
3.1.2 Where and How Well Can Disambiguation Succeed? 23
3.1.3 Why previous approaches are not appropriate for VLIW 25
3.2 Our Memory Disambiguation System 27
3.2.1 Overview 27
3.2.1.1 Interface and Interaction Protocol. 27
3.2.2 The Memory Disambiguation System and its Implementation 29
3.2.2.1 Flow graph builder. 29
3.2.2.2 Modified reach analyzer 30

3.2.2.3 Conventional constant folder 31
3.2.2.4 Generalized constant folder. 31
3.2.2.5 Variable folder 32
3.2.2.6 Loop analyzer 34
3.2.2.7 Algebraic expression normalizer. 36
3.2.2.8 Simple range analyzer 36
3.2.2.9 Memory Disambiguation proper. 37
3.2.2.10 Modified live-dead analyzer 38
3.2.3 An Example. 39
3.3 Problems with the Dynamic Interaction with Trace Scheduling 41
3.3.1 Reaching Definitions. 42
3.3.2 Live-dead Analysis. 42
3.4 Possible Extensions 43

Chapter 4: Correctness of Trace Scheduling 46
4.1 Preliminary Definitions 47
4.2 The Model — A Precise Description of the Trace Scheduler (TS) 50
4.2.1 Trace Picker (TP):. 50
4.2.2 The Compactor 50
4.2.2.1 Trace Compactor (TC): 51
4.2.2.2 Split Compensation (SC):. 51
4.2.2.3 Rejoin Compensation - 1 (RC1): 51
4.2.2.4 Rejoin Compensation - 2 (RC2): 52
4.2.2.5 Rejoin Compensation - 3 (RC3): 53
4.2.2.6 Continuation Resetting (CR):. 53
4.2.3 Some Observations 54
4.3 What Exactly Do We Want to Show? 55
4.4 Partial Correctness 56
4.4.1 Strategy 56
4.4.2 Partial Correctness Proof. 57
4.5 Termination of Trace Scheduling 69
4.5.1 Preliminaries 70
4.5.1.1 Observations 70
4.5.1.2 Definitions 71
4.5.2 Proof of the Termination of Trace Scheduling 72
4.5.2.1 Outline 72
4.5.2.2 Termination Proof proper 73
4.6 Windfall Profits Resulting form the Above. 82

Chapter 5: Flow Analysis, Disambiguation and Trace scheduling 83
5.1 Reach Analysis for Trace Scheduling 84
5.2 Live-Dead Analysis for Trace Scheduling 88
5.3 Live-Dead Analysis for Finer Granularity of Array References 94
5.3.1 The Problem 94
5.3.2 Our Solution 96
5.3.3 Implementation 97

Chapter 6: Runtime Disambiguation 99
- 6.1 The Idea of Runtime Disambiguation 99
- 6.2 Practical Concerns. 100
  - 6.2.1 Renaming. 100
  - 6.2.2 Correctness Considerations 101
  - 6.2.3 Space Considerations. 104
    - 6.2.3.1 RTD as Dynamic Assertions 105
    - 6.2.3.2 Smart Compiler Generated RTD 105
    - 6.2.3.3 Standard Optimizations 106
    - 6.2.3.4 RTD Code Elimination. 107
- 6.3 Implementation 109
  - 6.3.1 Modifying BULLDOG 109
  - 6.3.2 Correctness and Space Considerations 110
  - 6.3.3 Experience with RTD 111

Chapter 7: Conclusions and Future Work 113
- 7.1 Conclusions 113
- 7.2 Directions for Further Research 115
  - 7.2.1 Beyond the ELI 115

Appendix I: The Parafrase System 118
- I.1 Goals 118
- I.2 Outline of the System 118
- I.3 Frontend 119
- I.4 Intermediate. 119
- I.5 Backend 119
- I.6 Data-Dependence Analysis 120
- I.7 Methods Used for Reducing Dependencies 120
- I.8 Tree Height Reduction 121
- I.9 Recurrence Systems 121
- I.10 IF Nodes in Dependency Graphs 123
- I.11 IF Statements In Loops 123

Appendix II: The Memory Anti-Aliasing System 125
- II.1 The Interface 125
- II.2 Block Definition and Flow-Graph Builder 126
- II.3 The Interface Implementors 129
- II.4 Memory Reference Comparator. 133
- II.5 Solution to Diophantine Equation 135
- II.6 Generalized Constant and Variable Folding 137
- II.7 Loop Finder 140
- II.8 Simplifier/Canonizer 142

Appendix III: The RunTime Disambiguation System 146

List of Figures

Figure 1-1:	Trace scheduling loop free code. (a) A flow graph with each block representing a basic block of code. (b) A trace picked from the flow graph (c) The trace has been scheduled but not rejoined to the rest of the code. (d) The sections of unscheduled code that allow rejoining.	6
Figure 1-2:	The Structure of the Bulldog Compiler.	8
Figure 1-3:	Preliminary experiments: effects of disambiguation	9
Figure 3-1:	Disambiguation Example	26
Figure 3-2:	Disambiguation example	40
Figure 3-3:	Solution for the Diophantine Equation in Fig.3-2	41
Figure 3-4:	Reaching definitions:. Which point in (b) corresponds to p in (a) ? (a) Before compaction (b) After compaction	43
Figure 3-5:	Live-dead information changes: (a) Before compaction (b) After compaction	44
Figure 4-1:	A Program Graph	48
Figure 4-2:	A (Partially) Compacted Program Graph	48
Figure 4-3:	Trace Scheduling Dependency Examples	50
Figure 4-4:	Sample of code explosion resulting from RC1	52
Figure 4-5:	Trace-path S->Split-point	61
Figure 4-6:	Trace-path Rejoin-point->Exit	64
Figure 4-7:	Trace-path Rejoin-point->Split-point	66
Figure 4-8:	Definitions Examples	71
Figure 4-9:	Effect of TC on DAG	75
Figure 4-10:	Effect of SC and TC on DAG	77
Figure 4-11:	Effect of TC and RC on DAG	79
Figure 5-1:	Reaching definitions: (a) Before Compaction (b) After Compaction	85
Figure 5-2:	Reaching definitions are conservative (a) Before Compaction (b) After Compaction	87
Figure 5-3:	Live-dead information before compaction	89
Figure 5-4:	Live-dead information after compaction	89
Figure 5-5:	Live-dead Information During Compaction (a) Before Compaction (b) After Compaction: The effect of b' on live-dead information off-trace is nullified by b on trace.	91
Figure 5-6:	Incorrect Trace Compaction as a Result of Erroneous Live-dead Information) (a) Before compaction of trace S->E.	92
Figure 5-7:	Incorrect Trace Compaction as a Result of Erroneous Live-dead Information) (b) Before compaction of trace R->(*)->S1	93
Figure 5-8:	The Live-dead problem for Arrays in Trace Scheduling	96
Figure 6-1:	Original code before RTD code insertion	99
Figure 6-2:	Code after RTD code insertion	100

Figure 6-3: Impossible RTD code insertion 101
Figure 6-4: Sample problematic compactions using RTD 102
Figure 6-5: Sample compactions involving multiple *if*'s created by RTD 102
Figure 6-6: Requirements for Undoing Mechanism 104
Figure 6-7: Use of RTD as a dynamic assertion 105
Figure 6-8: Clustering of multiple RTD 106
Figure 6-9: Assertion tests unification 107
Figure 6-10: Eliminating copied RTD code from compensations 108
Figure 6-11: Preliminary results - RTD 111

List of Tables

Table 3-1: Effectiveness of Disambiguation in the BULLDOG compiler 24

Chapter 1

INTRODUCTION

1.1 Overview of This Thesis

Trace scheduling is a technique for transforming sequential programs into parallel programs for VLIW machines, a variety of tightly coupled multiprocessor. When this investigation began, trace scheduling was unimplemented and many serious questions of appropriateness and effectiveness needed to be solved. Chief among them was disambiguation, the act of determining at compile time whether two indirect references are to certainly different locations.

This thesis demonstrates that:

1. Disambiguation is practical.
2. Without disambiguation trace scheduling is much less effective.
3. Aspects of disambiguation which seemed possibly incorrect were in fact provably correct or easily modified to be so.

To demonstrate the practicality of disambiguation, a major implementation was undertaken. That implementation was part of the BULLDOG compiler for over a year, and carried out the techniques described in this thesis. The effectiveness and necessity of the disambiguator became overwhelmingly obvious as the trace scheduling compiler was built. Turning it off made the parallelism we found nearly disappear.

A trace scheduling compiler does many code motions that dramatically change the flow of control as code is generated. Disambiguation, and indeed trace scheduling, can't work correctly unless flow analysis information is constantly updated. However, doing a global, dynamic flow analysis is far too expensive. Unfortunately there are no intuitive reasons for believing that that is not required. It was therefore necessary to formally analyze our requirements. To do this, we had to put trace scheduling in a formal framework, prove that trace scheduling was itself correct, and then prove the necessary facts about flow analysis. Through this process we were able to show that incremental and local flow analysis is always correct for our purposes.

To understand how these assertions are demonstrated, the first three sections of this chapter provide background on: VLIW Architectures, Trace Scheduling and Memory Anti-aliasing, while the last two sections give a more detailed overview of this thesis and its role in the BULLDOG compiler and the ELI project in general.

In Chapter 2 of this thesis we discuss previous work related to this thesis, while in Chapter 3 the implementation of the disambiguator and our experiences with it are described in detail. Chapter 4 gives a formal definition of trace scheduling, together with a proof of correctness. Without these, we would have nothing with which to prove that incremental flow analysis does not destroy correctness. In chapter 5, that proof is given. Chapter 6 ponders future directions for disambiguation. In particular, a new technique, called Run Time Disambiguation is suggested and our implementation of this technique is described. Finally in Chapter 7 conclusions are presented together with directions for further research which have become apparent as a result of this work.

1.2 Very Long Instruction Word Architectures

A good description of VLIW architectures is found in [13]. In this section we will give a summary of that description.

The defining properties of VLIW Architectures are:

1. There is one central control unit issuing a single wide instruction per cycle.

2. Each wide instruction consists of many independent operations.
3. Each operation requires a small, statically predictable number of cycles to execute. Operations may be pipelined.

Restrictions 1 and 3 distinguish these from typical multiprocessor organizations.

Since it is nearly impossible to tightly couple very many highly complex operations, the underlying sequential architecture of a VLIW will invariably be that of a **Reduced Instruction Set Computer** or **RISC** [31].

VLIW machines might have large numbers of identical functional units. When they do, they are not required to be connected by some regular and concise scheme such as shuffles or cube connections. A tabular description of the somewhat ad hoc interconnections suffices for our purposes. This makes the use of VLIW machines very different from machines with regular interconnection structures and/or complex hardware data structures, as the irregularities of the architecture and the granularity preclude hand coding.

1.2.0.1 Why Haven't These Machines Been Built for Massive Parallelism?

Previously built machines whose architecture have the same flavor as VLIW's are horizontal microcoded engines, the CDC 6600 and its many successors, such as the scalar portion of the CRAY-1; the IBM Stretch and 360/91; and the Stanford MIPS [17]. However all of these machines have offered only a very limited amount of parallelism, due to the evidence that no large parallelism exists in basic blocks, and no good way of overcoming basic blocks restrictions existed when these machines were built.

Occasionally people have built more parallel VLIW machines (e.g. the AP-120, FPS-164 attached processors). But to obtain good speedups, these machines have to be hand coded at enormous cost, which may be acceptable only for very special purpose code. We're interested in much more parallelism; obviously hand coding is impractical.

We believe that our own experiments [29], coupled with the success the BULLDOG compiler has encountered so far (for a detailed description of this see [9]) demonstrate that the earlier pessimistic experimental measures should not be taken as upper bounds on available parallelism (more about this shortly).

Code generation for VLIW machines differs from the ordinary in that it does large scale code rearrangement in order to pack operations efficiently into wide instructions. If this process, called **compaction**, is not done, the code will usually be intolerably slow. We feel that **Trace Scheduling** [12], together with the techniques described here and in [9] and [33], enable effective VLIW machine compilers to be built.

1.3 Trace Scheduling

The gains from doing block by block compaction are very limited[1]. Therefore we must go beyond basic block boundaries if we are to efficiently exploit VLIW machines with large degrees of parallelism.

Trace Scheduling [10] is a technique for replacing the block-by-block compaction of code with the compaction of long streams of code, possibly thousands of instructions long. There are essentially four phases in the trace scheduling process.

First dynamic information about the flow of control of the program is obtained and is used at compile time to select "critical paths", that is streams with the highest probability of execution. While such dynamic information cannot always be obtained (e.g. for parsing loops or tree searches which are dominated by highly unpredictable jumps), one can make a good guess at the direction of most jumps in typical scientific code. These guesses may be derived by gathering statistics from actual runs of the uncompacted code on samples of data, or may be programmer supplied. While other heuristics could be used, this one has been found to give good results on the type of code we're interested in.

Once a trace has been selected for compaction, some preprocessing is performed on it so as to disallow code motions of operations that would clobber variables which are live off the trace. We consider such motions illegal and ensure their inhibition by creating "fake" data-dependencies between such operations and the appropriate conditional jumps.

---

[1]Basic blocks are quite small on the average and thus the speedups achievable from them is very small.

In the next phase, the scheduler is allowed to compact the whole trace as if it were one big basic block. The parallelism in the schedule thus obtained is only limited by the data-dependencies intrinsic in the trace and any resource conflicts which may arise; however basic block boundaries are effectively ignored.

After the trace has been compacted the scheduler will have made many code motions across conditional jumps. These code motions may well have altered the state of the program with respect to jumps into or out of the trace body. To restore correctness with respect to the outside world, some postprocessing is done. In this phase new code is inserted at the trace exits and entrances to recover the correct machine state outside the trace. Without this compensation pass, available parallelism would be essentially restricted to basic blocks by the need to preserve jump boundaries.

Once the postprocessing is over, one application of trace scheduling is complete. The next most frequently executed path is then selected for compaction, and the process repeats. This new trace may well include some or all of the compensation code generated as part of a previous trace compaction. When the repetition of this process reaches code with little probability of execution, we may choose, for space and compile time efficiency reasons, to use more mundane compaction methods.

Trace scheduling deals with loops by unwinding a number of iterations and compacting the resulting traces. All the intermediate loop tests will now be conditional jumps in traces; they require no special handling beyond that always given conditional jumps. In this context the effect of *software pipelining* is achieved as a natural byproduct. While this may be less space efficient than other techniques for automating software pipelining [27], [22], [23], it will be able to handle more general types of code.

A graphic illustration of the application of Trace Scheduling to a single trace is given in 1-1. A detailed, more formal definition of trace scheduling will be given later on in this thesis, in the correctness chapter.

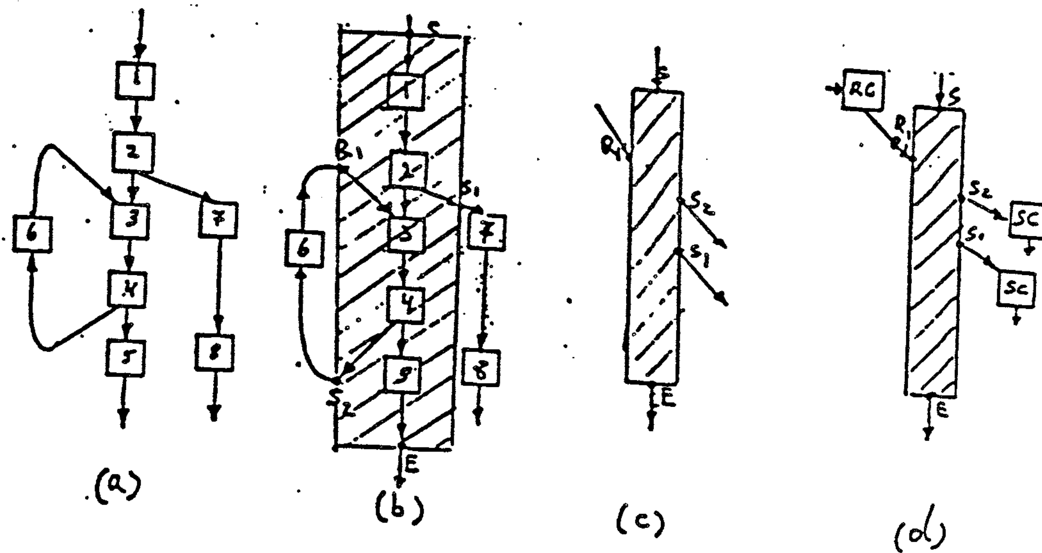


Figure 1-1: Trace scheduling loop free code.
(a) A flow graph with each block representing a basic block of code.
(b) A trace picked from the flow graph
(c) The trace has been scheduled but not rejoined to the rest of the code.
(d) The sections of unscheduled code that allow rejoining.

1.4 The Problem of Memory Anti-Aliasing

In the process of compacting a sequential program we need to do massive numbers of code motions to fill instructions with operations that come from far removed places in the program. Code motions are restricted - in order to preserve the semantics of the original program - by the data precedence relation. For example if our program has the steps:

```
(1)    Z := A + X
       ............
(2)    A := Y + W
```

it is important that our code motions don't cause (2) to be scheduled earlier than (1).

This is a straightforward matter. But what happens when A is an array reference?

```
(1)    Z          :=  A[expr1] + X
       ........................
(2)    A[expr2]   :=  Y + W
```

Whether (2) may be done earlier than (1) is ambiguous. If expr1 can be guaranteed to be different from expr2, then the code motion is legal, otherwise it is not. Answering this question is the problem of **Anti-Aliasing Memory References or Disambiguation**[2]. Being able to do this correctly at compile time is important for any compacting compiler, and particularly for those seeking to exploit very fine grained parallelism.

1.4.1 Where and How Well Can Disambiguation Succeed?

Some references, such as chasing pointers down a list or indirect references which depend on runtime information, cannot be disambiguated by any amount of compile time analysis. Thus for several important types of code, such as systems code which is heavily data driven or uses complex indirect references (e.g. compilers, data bases), even the most sophisticated disambiguation system won't do well. This type of code also has several other undesirable properties (e.g. unpredictable control flow) which make it an unlikely candidate for effective Trace Scheduling.

On the other hand, indirect references in inner loops of scientific code are mostly array references. 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 a Trace Scheduling compiler and VLIW machines, since too conservative an approach will lead to inefficient use of the machine (and small speed-ups)[3].

---

[2]Anti-aliasing is an overloaded term. We will generally use the word "disambiguation" in this thesis.

[3]In fact this intuition is supported by the results obtained in experimenting with the BULLDOG compiler.

1.5 Where Does This Work Fit in the Greater Scheme of Things?

1.5.1 Overview of the BULLDOG Compiler

The ELI group at Yale, has implemented a trace-scheduling compiler in compiled Maclisp on a DEC-2060. It is called Bulldog to suggest its tenacity. Bulldog has 5 major modules, as outlined in figure 1-2.

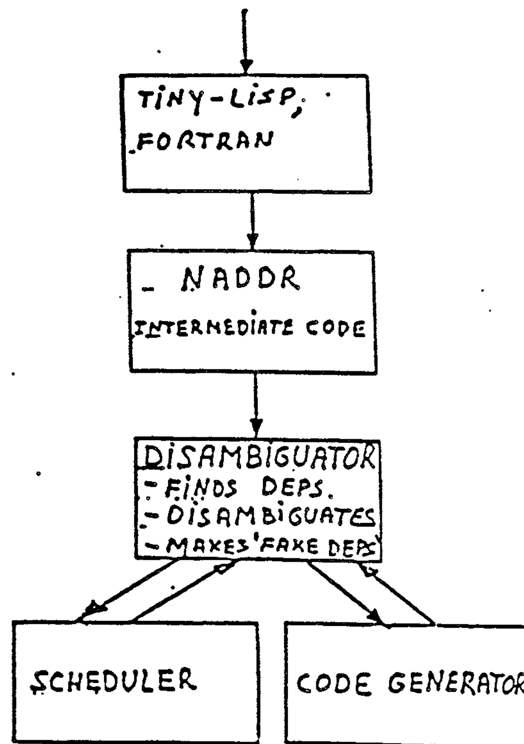


Figure 1-2: The Structure of the Bulldog Compiler.

Several code generators have been built: one for an idealized VLIW machine that takes a single cycle to execute each of its RISC-level operations and does an unlimited number of memory accesses per cycle. This code generator is used to help debug the other modules of the compiler and to measure potential parallelism exploitable by trace scheduling in conjunction with disambiguation. This is a primary tool for fine tuning the compiler.

The other two generators, are both for more realistic architectures (parametrized versions of the basic ELI and the FPS-164) and are used to tune the architecture to be implemented. They also provide a measure of the actual parallelism obtainable by Bulldog/VLIW approach. The results to date have been encoraging (e.g. more than an order of magnitude speedups for FFT's). An in depth discussion of the different strategies used by the ELI generators and a detailed description of the overall compiler can be found in [9] and [33].

The input to the compiler is a local Lisp-sugared Fortran, C, or Pascal level language called Tiny-Lisp. The front end generates RISC-level intermediate code which we call N-address code or NADDR. A FORTRAN '77 subset compiler into NADDR has also been written, and other languages may be considered after that. A major advantage of the RISC-level NADDR is that it is very easy to generate code for and to apply standard compiler optimizations to.

	TRANSITIVE CLOSURE			INVERSION (BUBBLE) SORT		
	(a) 3x3 Matrix, Diagonal (cycle graph) w/ mem.disambiguation	(b) 3x3 Matrix, Diagonal (cycle graph) w/o mem.disambiguation	(c) 9x9 Matrix, Diagonal (cycle graph) w/ mem.disambiguation	(d) Reversed Array Of Length 10 w/ Mem Disambiguation	(e) Reversed Array Of Length 10 w/o Mem Disambiguation	(f) Reversed Array Of Length 50 w/ Mem Disambiguation
STREAM LENGTH	425	425	6296	1308	1308	33483
MIN.SCHED LENGTH	8	17	14	28	93	149
MAX.LEVEL WIDTH	81	81	619	96	96	1378
TEMPS ADDED	302	302	4592	855	855	22498
NUMBER OF BLOCKS	101	101	1568	358	358	8879
AVG BLOCK LENGTH (BEFORE)	4.21	4.21	4.02	3.65	3.65	3.82
AVG BLOCK LENGTH (AFTER)	2.76	2.76	2.72	1.34	1.34	1.42
SPEED-UP PER BLOCK	1.52	1.52	1.48	2.72	2.72	2.69
GLOBAL SPEED-UP	53.13	25.00	449.7	46.71	14.06	227.4

Figure 1-3: Preliminary experiments: effects of disambiguation

1.5.2 The Role of this Thesis in the ELI Project

As part of the ELI effort, we designed and built a disambiguator module which answers the requirements for "interactive" fine grained dependency prediction and disambiguation of the trace scheduler and code generator modules. We did not spend a lot of time investigating the need for such a system, though some preliminary experiments (see figure 1-3) supported our intuition. We believed that the need is obvious, particularly in the context of fine grain parallelism exploitation from code in which a major source of potential parallelism lays in innermost loops involving indirect memory references. Since this is precisely the kind of code we expect our compiler to work on, we thought that disambiguation will be crucial to the success of a trace scheduling compiler. In fact, experience with the BULDOG compiler to date overwhelmingly supports this view. As will be shown in (section 3.1.2), the parallelism found by the compiler for array code is drastically reduced when disambiguation is turned off. We had to undertake a formal analysis of the interaction between the disambiguation system and the (trace) scheduler and code generator modules, in order to ensure the correctness of the disambiguator's predictions which in turn depend on the flow information of the program. Furthermore, the disambiguator is also responsible for restraining the trace scheduler from making absolutely illegal between the blocks code motions (the "preprocessing" described in section), which also requires accurate flow information. The arrows going back and forth from the disambiguator to the scheduler and code generator modules in figure 1-2 attempt to emphasize the interactive nature of this communication. It is this dynamic interaction between the modules and its ensuing effect on flow information that defied our intuition, and led us to formalize Trace Scheduling and prove its correctness. The mechanisms thus developed enabled us to precisely define and analyze our flow information requirements and provide correct solutions.

The last part of this thesis, the run-time disambiguation system, is a natural extension of the notion of memory disambiguation, for references which cannot be disambiguated at compile time. While implemented, it is not part of the BULDOG compiler, and thus does not appear in figure 1-2. While the preliminary results we obtained in experimenting with

this system are encouraging, more testing and analysis are required to fine tune the best approach to this latter problem.

Chapter 2

PREVIOUS WORK

2.1 Introduction

In this chapter we will review related work in the field of compilation for very parallel architectures. In particular, we will describe early experimental work which influenced previous research, the architectures which resulted and approaches used in exploiting parallelism given these architectures. Finally we will outline the differences between our work and these approaches.

Before the advent of trace scheduling, VLIW machines could not be effectively hand coded and in the absence of good global compaction techniques they were regarded as only suited for special purpose, limited parallelism applications. As a result, previous work done in the field of compilation of ordinary programs for execution on very parallel machines was centered around vector, array and pipeline processors as well as multi-processors. A major effort in this domain was undertaken at the University of Illinois at Urbana, where D. Kuck and his group developed a system called Parafrase, whose main goal is to generate code for fast "highly parallel" machines.

2.2 What Were the Earlier Experiments?

Tjaden and Flynn [34] and Foster and Riseman [14] did the following:

- They took actual machine language execution streams from normal programs running on various CDC/IBM machines.
- They broke the streams up into basic blocks.
- They asked: given infinite hardware (they were thinking in terms of an infinite number of CDC/IBM functional units), how much faster would the programs have been executed. We may assume that instruction issue is instantaneous.

When one considers the hardware that inspired these experiments, this is a quite reasonable question to ask. These machines use run time hardware scheduling, assigning operations to functional units, to take advantage of fine grained parallelism. The hardware really can't know what to do in the face of a conditional jump. It has to play safe, not being able to do a data flow analysis of the program, and it has to wait until the conditional jump settles before writing variables with possibly incorrect values.

Various machines, including the IBM STRETCH and 360/91, attempted some amount of calculation past jumps, but they didn't do anything irreversible until the jump settled. This seemed a limited benefit.

But what would one do if one *really* had infinite hardware? One would have been running the same program on two identical machines, and whenever a conditional jump came along, one would let the two of them take different paths. One would then just throw out the wrong one when the test settled. In a companion to one of the above experiments, that is exactly what was measured [32]. Though they found extensive speed-ups might be available, they noted that obtaining even a ten-fold speed-up mandated that one would have to start with $O(2^{16})$ machines, a rather unacceptable hardware cost. Thus exploiting parallelism beyond basic blocks was ruled out as impractical.

2.3 Other Previous Experiments

Many suggestions for improved parallelism location have been reported in the literature, sometimes accompanied by empirical measures of effectiveness. Important work done by David Kuck's group at The University of Illinois [22], [5], [3] has measured the actual parallelism obtainable by such algorithm transformations as recurrence solution and tree height reduction.

All of these were quite different from our goal of obtaining an upper bound on the *potential* parallelism achievable from the parallel execution of ordinary programs.

2.4 Our Experiment

To establish the applicability of trace scheduling techniques to the compaction of ordinary (scientific) code for parallel execution, we undertook an experiment which measured the potential parallelism available in such programs, under idealized trace scheduling conditions. For a more detailed description of the experiment see [29]).

2.4.1 So What Was Wrong With the Earlier Experiments?

There are several problems with the above experiments. To start with, they were all heavily biased towards hardware oriented schemes of identifying parallelism. As such, they spent a great deal of effort in adapting their model to closely emulate the idiosyncrasies of the particular hardware which motivated their experiments, reducing the generality of the results as measures of parallelism available in ordinary programs.

Even more important, this approach effectively limited the measured parallelism to basic blocks. *Dynamically* overcoming basic block boundaries on a large scale was deemed completely unrealistic and they did not foresee any possibility of bypassing large numbers of conditional jumps *statically* at compile time. Given the unavailability of good global compaction methods at the time, this attitude was quite natural.

The domain from which the sample programs were taken further reduces the utility of these results for our purposes. All experiments dealt with seemingly randomly chosen

programs from several applications areas. Most of them appear to be heavily data-driven (e.g. compiler, sorter, string matching), some with complex flow of control. Such algorithms are not likely to be amenable to effective compile time global compaction techniques, and thus are of limited interest from our perspective. Furthermore the data-dependent nature of control flow in these programs negatively affects the robustness of the results.

2.4.2 Trace Scheduling Conditional Jumps and Ambiguous References

People who code VLIW architectures would not consider restricting themselves to basic blocks while doing code rearrangements. It is almost always possible to move an operation up past a jump, making one branch shorter at the expense of the other. Similarly, operations may move down into one or both branches. These operations may be done "for free" in unused fields of the target instruction. As we have seen in the previous chapter, this is precisely what trace scheduling attempts to do. It uses jump predictions to pick (hopefully critical) traces which it then compacts as single blocks, inserting compensation code when necessary.

Of course, the efficiency of this process depends on the accuracy of the predictions and thus will be limited to code which is not dominated by highly unpredictable jumps. Similarly, any static (compile time) scheduler will be hindered in its work by ambiguous memory references (to be discussed shortly). While some array references lend themselves well to compile time disambiguation, others such as pointers are usually not amenable to such techniques.

These were important issues which had to be considered in choosing an appropriate domain for our experiment. Parallelism available in programs exhibiting highly data-driven control flow and intractable references, however large, would be irrelevant (since unexploitable) for our purposes.

2.4.3 Using an Oracle to Measure Available Parallelism

Given trace scheduling, it is clear that the restriction to basic blocks is not realistic. How may we obtain a more reasonable empirical upper bound on what we could do with very cheap hardware? A reasonable attack is to assume that an **oracle** is present to guide us at every conditional jump, telling us which way the jump will go each time, as well as disambiguating with 100% accuracy indirect memory references. This is not so unreasonable, given that trace scheduling uses guesses at this same information, and that most scientific code (our target domain) is quite static. Similarly, our disambiguation mechanism attempts to disambiguate indirect references, and when these are relatively simple array references, (as in large numbers of programs in our sample domain) a sophisticated system can do quite well.

2.4.4 Results

2.4.4.1 Global Speed-up Ratio

Once we overcame the bottlenecks that conditional jumps and ambiguous references impose on parallelism, we found that *the available global speed-ups ranged from below 4 to 988 times increase in execution speed.*

Many of the programs were limited only by the size of their data.[4] If anything, the data we picked was smaller than that encountered in realistic applications, since we had to simulate the code.

Several of these programs are rather sequential in nature. Thus one would expect less potential parallelism, and that is the case. This is probably parallelism that a compacting compiler could take advantage of, though, since it does not generally involve disambiguating memory references.

The programs that exhibit the largest parallelism operate on arrays. Intuitively, array processing presents greater opportunities for parallelism, and these results bear this out.

---

[4]While the program streams were not influenced greatly by the data contents, size made a difference in the amount of parallelism available.

The magnitude of the speed-ups clearly indicates that they do not result from the parallelization of trivial parts of the programs (such as the initialization of arrays), but rather that the parallelism is intrinsic to the whole computation process. Here the disambiguation of memory references is particularly important, so it is more problematical whether practical compaction methods will be able to approach the upper bound. However, the amount of available parallelism is large enough to encourage the development of sophisticated (and costly) disambiguation techniques.

2.4.4.2 Basic Block Compaction

We found that when compaction was limited to basic blocks only, the available speed-up was almost always less than a factor of 2.5. This is consistent with earlier experiments. The difference between this measure and the global speed-up is dramatic enough by itself to account for a large measure of the pessimism that has been felt about VLIW architectures. If trace scheduling, or some other global compaction technique, can recover a large share of this difference, then there is real hope for using VLIW architectures for significant parallelism.

2.5 Compiling ordinary code for parallel execution: Parafrase

The major effort in the domain of compilation for parallel machines was undertaken at the University of Illinois at Urbana, where D. Kuck and his group developed a system called Parafrase, whose main goal is to generate code for fast "highly parallel" machines [24], [23].

Parafrase relies on extensive (and expensive) global data-dependence analysis (Pi-Partitioning) and essentially no global flow information. A memory disambiguation mechanism, developed by Banerjee [4], is used to reduce superfluous dependency edges (due to ambiguous array references) in building the data-dependency graph. A partitioning algorithm identifies maximal sets of blocks cyclically connected by dependencies, called pi-blocks. These blocks are then partitioned into groups in which each member can be executed independently from the others, and all such groups are scheduled in dependency

order.

Once the input program is partitioned, the system uses a series of general (e.g. Renaming, Loop Fusion) and special purpose (e.g. Expansion, Recurrence Solving) **source level** transformations in order to enhance the potential for finding parallelism. Since this parallelism has to be very structured, Parafrase devotes a great deal of effort (e.g. loop blocking, IF-Transformations) to trying to make the parallelism available in the program fit that of the target architectures. In the next section we will dwell on the approach that Parafrase uses in finding dependencies and performing disambiguation. For completeness, an overall description of Parafrase and its major techniques for parallelism exploitation are given in appendix I.

2.5.1 Construction of the Dependence Graph and Disambiguation

This is where the disambiguation really comes into play in the Parafrase system. The techniques used have been introduced in Banerjee's thesis [4]. Other methods such as [25], [36], [20], are enhancements of the original approach or modifications, or are dealing with very specific problems, (e.g. vectorization of particular types of loops). An approach more similar in flavor to the one we use is hinted at in a paper by Heuft and Little [18]. They point out, as a critique of a previous paper by Banerjee [3], that taking into account the dynamic behavior of a loop may allow very different compaction (i.e. better) than the one suggested by Banerjee. Since trace scheduling attempts to achieve precisely such dynamic effects at compile time, the spirit of that paper is closer to that of the disambiguator presented in this thesis. Unfortunately, the paper only deals with a particular example of a certain type of loop, and no algorithmic solution is provided for even that case.

The method that Banerjee's system uses for disambiguation is the solution of diophantine equations, in 2 or more unknowns, which was also suggested by Lamport and others. The algorithms offered are essentially the classical number theory ones with some enhancements to allow range constraints. There is no variable folding, or fine grained analysis nor is there any extensive support for the refinement of disambiguation results. Since Parafrase exploits

parallelism mainly by vectorizing and recurrence solving, disambiguation is a less important task than in our system. In this light it does seem that disambiguation in Parafrase is more coarse and thus does not need the extra precision obtainable by the use of global flow and range analysis and other methods which can be used to enhance disambiguation. (An extensive study of range analysis has been undertaken by W. Harrison [15]).

The functioning of the disambiguation system in Parafrase can be summarized as follows:

- Use mode vectors to handle conditionals, by realizing that 2 references which are on different paths from a conditional statement can't collide.
- Use loop bounds to refine results.
- Simple index variable elimination.
- The disambiguation proper consists only of a diophantine equations solver on the indexes of 2 array references, which are assumed to share at least some encompassing loops[5].

Given the concern with vectorization, the solutions that the system must give the equations are in more unknowns than strictly necessary. This results in longer computations and coarser results (all or nothing).

For example for a situation like:

```
Do i1 = 1,n
Do i2 = 1,m
 ref1{i1,i2}
 .......
 ref2{i1,i2}
```

The system would have to solve

$$a_2*x_1 + a_1*x_2 + a_0 = b_2*x_3 + b_1*x_4 + b_0 \text{ (4 variables)}$$

presumably since they are interested in knowing whether vectorizing is feasible, whereas only

$$a'_2*x_1 + a'_1*x_2 + a'_0 = b'_2*x_1 + b'_1*x_2 + b'_0 \text{ (2 variables)}$$

[5]Given the target architectures, this assumption is natural.

needs to be solved when interested in particular iterations.

The disambiguator is also used in the cycle breaking algorithm of Parafrase. The idea is to divide a loop into smaller separate loops with smaller bounds such that we isolate the dependent parts some of which may be done in parallel (i.e. vectorized). Again because of the system's orientation, the statements have to be broken completely apart, creating one statement loops which may be turned into vector operations. As a result the algorithm is more complex than need be (see below).

2.6 Differences in our approach

Our techniques differ from those of Parafrase in several important aspects. First and foremost, we are dealing with very fine grained parallelism, on VLIW machines, which Trace Scheduling makes effective. As a result a large number of special case transformations (e.g. expansion, tree-height reduction, loop-blocking, and most important, special conditional handling) become superfluous in our approach, since equivalent results are achievable by trace scheduling coupled with (smart) unwinding and memory disambiguation, at a finer level of granularity. Still, some of the source level transformations in Parafrase (e.g. reccurence solving) would be useful, and could be incorporated in our compilers.

We don't use complete data-flow graphs, but dynamically updated relevant parts. This is a big efficiency gain for large programs, given the low granularity level involved, as it would be too expensive to keep full dependency graphs in memory and update them dynamically during trace scheduling. We use a simpler data-dependency analysis on a trace by trace basis, coupled with sophisticated flow-analysis to achieve the accuracy required for a very discriminating disambiguation mechanism, which is crucial for the maximum parallelism to be exploited. This disambiguation mechanism, coupled with loop unwinding achieves better performance and more generality.

Because of the above, and the fact that the disambiguator in our system also has to dynamically create fake dependencies to implement the pre-processing phase 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 throughout the compaction process.

2.7 Other Related Work

In the above we have outlined previous techniques for exploiting parallelism in ordinary sequential programs and have described early experiments which were partly responsible for the neglect of VLIW architectures for large parallelism. 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 in this thesis, should also be mentioned.

A more radical approach, that of **dataflow/reduction models** developed primarily at MIT [8], UC Irvine [2], Utah [19], [7],and in France [6], 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, such as Treleaven's [35].

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 [26] 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.

Chapter 3

MEMORY DISAMBIGUATION

The disambiguation system is vital to the successful operation of a Trace Scheduling compiler (BULLDOG). This claim is based on the limited preliminary experiments we undertook, on our understanding of the needs of a compactor which deals with functional unit level parallelism and on actual experience with the BULLDOG compiler. Such results will be shortly discussed in Section 3.1.2.

To attempt to disambiguate as many ambiguous references as possible, we have implemented a disambiguation system which relies on conventional data-flow analysis methods (e.g. reach and live-dead analysis) and unconventional ones (e.g. variable folding and special loop analysis). These techniques gather all the information we can find at compile time about ambiguous memory references, which are then compared, (e.g. by solving diophantine equations). Range analysis, if needed, is used to further refine the disambiguation results.

3.1 Background

3.1.1 Why do we need disambiguation?

To take advantage of the parallelism made available by Trace Scheduling, it is necessary to do massive numbers of code motions and fill instructions with operations that come from widely separated places in the program. Since the compiler has to deal with relatively low level operations on an operation by operation basis, very accurate prediction of data-dependencies is crucial to its success and that of VLIW machines, since too conservative an approach will constitute a major bottleneck for parallelism and thus lead to inefficient use of the machine (and small speed-ups)[6]. On the other hand, too liberal an approach may lead to illegal code movement and violate the correctness of the resulting program.

The fact that the parallelism we seek to exploit is at such a fine-granularity level makes the above problem especially acute. Indeed, we will see shortly that previous approaches, which aimed at coarser parallelism exploitation, are not adequate for our needs.

3.1.2 Where and How Well Can Disambiguation Succeed?

With complex forms of indirection, such as chasing down pointers, there is little hope of success. This is so since pointer references are too often too dependent on dynamic runtime data for a compile time comparison to be of any use. On the other hand, when indirect references are to array elements, we can usually disambiguate them.

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. Our system 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.

Indeed evidence supporting both the effectiveness of disambiguation and its importance

---

[6]For an illustration of this negative effect, see Figure 1-3 of chapter 1.

Program	Compacted Instructions Produced W/O Disambiguation	With Disambiguation
LN1	92	92
LN20	35	35
SQRT1	24	24
SQRT10	6	6
DOTPROD	203	203
FFT1	1215	1135
FFT4	869	521
FFT16	800	415
SOLVE1	10007	10007
SOLVE4	8625	5169
TRID1	2988	2988
TRID8	2469	1401
MATMUL1	109	109
MATMUL4	53	41
PRIME1	656	656
PRIME4	427	321
TRCL	78	53

Table 3-1: Effectiveness of Disambiguation in the BULLDOG compiler

for a trace scheduling compiler in particular is provided by our experiments with the BULLDOG compiler. In table 3-1 we compare the results obtained by the BULLDOG compiler with and without the disambiguator system, for several programs taken from our usual testbed. Even with the limited unwinding used for these tests, the importance of disambiguation becomes obvious. Larger unwindings are to be expected in real applications. This would further increase the significance of disambiguation for the performance of the compiler.

The first group of programs (LN, SQRT, MAX) don't contain any array references and thus are not affected by disambiguation. The speedups obtained from such programs are rather small, however. Any speedups achieved are a result of the software pipelining effect which trace scheduling achieves uniformly and naturally.

The second group (DOTPROD) involves indirect references. However, there are no ambiguities involved between them in any case, and thus the disambiguator again does not affect the compaction. Together with the previous programs these form a control group for this discussion.

The more interesting programs which occur in the following groups (FFT, SOLVE, TRID, MATMUL, PRIME, TRCL) are all dramatically improved by the use of the disambiguation system; the speedup is essentially doubled in several cases by the disambiguation. These programs are in fact typical of the target domain for our compiler. As expected, the improvement is particularly large when the traces are long and the potential speedups obtainable by trace scheduling are relatively large. This happens when the important (innermost) loops are unwound. When unwinding is not done, or traces are still small, the length of the compacted schedule will again be dominated by simple arithmetic (e.g. index calculations) and no large speedups will be achievable in any case. Under these conditions the effect of disambiguation decreases.

Of particular interest is TRCL. Even though trace scheduling will not perform too well on this program (due to its data-dependent nature), and the further handicap created by the tiny data size used in our experiment (a 3x3 matrix), the speedup achieved by the use of disambiguation is quite dramatic. This emphasizes our claim that in cases where potential parallelism exists beyond that achievable from software pipelining, and indirect references are involved, disambiguation will be crucial in effectively exploiting this parallelism.

3.1.3 Why previous approaches are not appropriate for VLIW

As we mentioned before, the only documented implementation of a Memory Disambiguation system that we are aware of, is due to Banerjee as part of the Parafrase compiler. While several other techniques, in the work of Lamport and Kuck, have an indirect bearing on the subject[7], none of them provide a satisfactory solution to our

[7] For example the hyperplane method, when successful would not require disambiguation of the type we describe here.

problem.

To start with, all the previously developed systems have one thing in common: their all or nothing approach, which for us is out of the question. For example, in figure 3-1 it would be unacceptable for us to decide that (a) and (b) cannot be scheduled independent of each other just because there may be a conflict between some iterations.

```
          for I := 2 to N do
(a)    A(I) := A(I-1)+ tot;
(b)    A(I+2) := A(I+1) + tot;
            tot := tot + B(I)
            end
```

Figure 3-1: Disambiguation Example

We believe that the previous techniques may be useful in a broader context, but by themselves they will produce sufficient parallelism only in a small percentage of programs. Because of the fine-granularity level of the parallelism we hope to exploit we can do much better if only we can disambiguate memory references accurately at an operation by operation level. To do this we need a more general approach which works uniformly on all code. In this respect, the lack of extensive global flow analysis information and the use of special purpose handling of conditionals in previous systems is particularly inappropriate for our needs.

Even more important is the fact that if our disambiguator is to be at all useful, it must be an integral part of the compiler, not just another aid. Thus it cannot be statically called in the process of building the data-dependency graph of the program. As a mater of fact, a full data-dependency graph at the level of granularity we are interested in is likely to be too large to be built in one piece. Furthermore, continuously updating such a graph with the changes introduced by the trace scheduler would be extremely inefficient. So the disambiguation must be a truly dynamic process, constantly interacting with the Trace Scheduler. Using this approach, the disambiguator needs only concern itself with one trace at a time, and deals only with simple dependencies (no cycles).

As a result of the interaction with the trace scheduler, several hard problems faced by other systems become trivial to overcome, as for example the handling of conditionals. On the other hand, the global changes introduced by the Trace Scheduler are so extensive, that the disambiguations system must keep close track of the scheduler's actions and even dynamically compute and update its information according to the current focus of attention of the Trace Scheduler. As we will see in the following chapters, this is not only crucial for accuracy, but also for the correctness of the compaction process.

3.2 Our Memory Disambiguation System

3.2.1 Overview

The system implemented for the Bulldog compiler compares any given references and tries to establish whether they do or do not refer to the same memory location. This is trivial in the case of scalar variables. When array references are compared, the disambiguator attempts to solve the equation expr-index1 = expr-index2. It uses **Reaching Definitions** [1] to narrow the range of each variable in the expressions. Assuming that the variables involved are integers, we use a diophantine equation solver to determine whether or not the two could be the same. Range analysis can be quite sophisticated. In the implemented system, definitions are propagated as far as possible, and equations are solved in terms of simplest variables possible. The system was implemented in DEC-20 MACLISP which was then compiled. It consists of about 3000 lines of code. The following is an in depth discussion of the memory disambiguation system. Details of the actual implementation of the more interesting modules can be found in Appendix II.

3.2.1.1 Interface and Interaction Protocol

The disambiguator is a main part of the BULLDOG compiler; it must work at a fine level of granularity and supply information to the trace scheduler and/or code generators on an operation by operation basis. It computes the relevant parts of the dependency graph dynamically, in response to the actions and requests of the trace scheduler. The

following is a short description of the memory disambiguation mechanism as it appears to the outside world (i.e.the trace picker and the code-generator)[8].

The trace picker picks out individual traces from the NADDR program and hands the traces one at a time to the code-generator. The code-generator treats the trace almost like a basic block, building a schedule for it. In order to do its work the code-generator must know which operands refer, or might refer, to the same locations. To get this information the code-generator presents the source operations of a trace one by one to the disambiguator, and the disambiguator replies with which operations, in the current trace, are data predecessors and the reason why they are predecessors.

Obviously, in the case of uncertainty, the code-generator should receive conservative answers; every time it is told operands reference different locations, it should be 100% sure. If the disambiguator isn't sure, then it assumes two operands may reference the same location.

Initially the disambiguator is primed by a call to **(NEW-PROGRAM PROGRAM)**. This is initiated by the trace scheduler when beginning to process the new NADDR program, and enables the disambiguator to set up its data-structures and perform some initial static analysis on the program (Reach, original live-dead, constant/variable folding and loop-analysis are performed at this point). When the code generator is then ready to process a new trace, it activates the disambiguator with a **(START-TRACE)** call. This signals to the disambiguator to purge its old information and get ready to start processing the next trace.

Further interaction in the same trace takes place through calls to **(PREDECESSORS SOURCE-OPERATION TRACE-DIRECTION PTR)**. This presents each successive source operation from the trace to the disambiguator. PTR is meaningful only to the code-generator (it is a pointer in its data structures). The disambiguator simply stores it and later returns it to signify that this operation is a predecessor. TRACE-DIRECTION is

[8]The interface protocol was defined by John Ellis, who is responsible for the overall design of the Bulldog compiler.

meaningful only if SOURCE-OPERATION is a conditional jump, and tells which way the jump will go on the trace. This is used by the disambiguator in establishing preemptive data-dependencies on conditional jumps to ensure the correctness of the trace scheduling transformations (see next chapter). Upon returning, PREDECESSORS returns the list of all previous operations on the trace that might be data predecessors of this operation, why they are data dependent and what causes the dependency. For a more detailed description of the interface, see Appendix II.1.

To keep the interface as simple as possible, the NADDR source program is represented as a list of NADDR source operations, where each NADDR source operation is an atomic object — that is, EQ tests will be used to determine equality of two source operations. This lets us represent source operations just as Lisp pointers. The disambiguator uses a hash table to map these pointers or list structures onto other information (Maclisp supports very efficient hashing). Because of the size of the data it deals with (programs of more than 1000 operations) and the intensive nature of the interaction, the disambiguator must use efficient algorithms. In particular it would not suffice to use linear searches to map the source operations onto internal data structures. Details of the implementation can be found in Appendix II.3.

3.2.2 The Memory Disambiguation System and its Implementation

The system consists of the following main modules:

3.2.2.1 Flow graph builder

This module is activated by the call to NEW-PROGRAM. It breaks the input program into basic blocks and constructs the program flow graph, using conventional techniques. Basic blocks are identified by first finding the leaders - statements that are either the first in the program or are the target of a conditional/unconditional jump[9]. The building of the program graph proper is essentially a depth-first search, with new blocks and edges added

[9]Since NADDR conditional jumps always have two explicit branches and no fall-through, these are the only positions where a leader can occur.

to the structure as they are found. For efficiency reasons the program itself is only partly recursive, and partly iterative (for going through the unconditional control transfers).

The structures used in the graph construction are common to all the disambiguator modules and store all the information required in the disambiguation process. In particular, this graph is used for the flow analysis that follows. To improve the efficiency of the analysis algorithms, the list of basic blocks produced is sorted in depth first order [1]. The depth first ordering is also used by the loop analyzer. For the upper level of this code, as well as a detailed description of the data-structures used, see Appendix II.2

3.2.2.2 Modified reach analyzer

Once the flow-graph is built and the basic blocks are sorted, reach analysis can proceed. This reach analysis is used as the basis for the reduction of index variables to a canonical form consisting of only constants, standardized loop counters and irreducible variables. The analysis itself is done using the well known [1] data-flow formulas:

$$OUT_n = IN_n - [KILL_n \cup GEN_n]$$
$$IN_n = \mathrm{UNION}_{p \in predecessors(n)} OUT_p$$

The computation starts with the assumption that nothing reaches statement n, and then iteratively obtaining a better approximation, by recomputing IN_n and OUT_n, until IN_n does not change anymore.

While the above is sufficient to obtain correct results, There are various modifications that we introduced to improve efficiency. Except for the first pass, information is computed on a block by block basis, and only stored at the leaders of each block. Furthermore we store, rather than recompute each time, information local to each basic block, i.e. OUT, GEN and KILL. We update OUT only for the blocks where it actually changes. Since this will be most efficient when the nodes are visited in depth first order (the block executed first given first), we use our depth first sorting to obtain the desired list of nodes. These changes achieve quite a dramatic improvement in the performance of the reach analyzer.

Once the results settle, we propagate them to the places where they may be needed for disambiguation (at statements using indirect memory references). In doing so we preserve

only information needed for our purpose. In particular, once the preliminary analysis phase ends, only definitions for variables used to index indirect memory references and their derivations are preserved. Later on in this chapter, and in Chapter 5, we will discuss the correctness of this approach and ways in which it can be modified to improve accuracy, in the presence of the trace scheduling transformations.

3.2.2.3 Conventional constant folder

The constant folding is performed on the actual operations and is thus incorporated in the generated parallel code. When only one definition reaches a use, and it is equal to a constant, or a constant expression, the use is changed to that value. Then if the statement can be folded (i.e. if its operation can be performed, because all its operands are now constants), it is replaced by the results of its evaluation. Then this new definition is propagated, and so on, until no more changes can occur. Again, little tricks help increase the efficiency of the code. As before we use depth first ordering to speed the settling of the computation. In addition, by propagating actual pointers to the definitions (rater than copies), we get free propagation of changes every time a statement is changed. This is a tremendous gain, not only in speeding up the constant folding and eliminating consing, but also in avoiding the need to communicate the changes introduced by this module to the rest of the compiler (i.e. trace scheduler and code generator). Notice that the constant folding is of the usual variety [1], which cannot proceed with the folding when more than one use can reach a definition. Worth mentioning is the fact that array indexes are in no way special and are folded — when possible — like any other variable.

3.2.2.4 Generalized constant folder

This is the logical (illogical ?) extension to the above: the folding goes on despite multiple reaching definitions (we have multiple possible values - a list of values - for such a variable). This allows the process to continue past jumps and rejoins. We should note that unlike the regular constant folding this phase cannot change the actual NADDR code, since it is only a compilation aid and as such has no real counterpart in the code generated (for example there is no way we could do multiple value operations in the current NADDR).

Thus the information gathered by the generalized constant folder is manifest only in the IN (reaching) definitions of every block. These definitions are now stored in a more compact representation which allows multiple values to be manipulated with relative ease. This is critical since the operations to be done on these structures are relatively complex: simplification, comparison, insertion, deletion, and general arithmetic operations. Furthermore, since we are now dealing with multiple values, represented as lists (which may even be nested), all the above operations must be generalized as well, to correctly apply to these structures.

An advantage of the fact that this pass comes after the normal reach analysis (and after the regular constant folding - which also changes the in/out definitions of every block), is that the information needed is *locally available*. Furthermore, the use of destructive operations gives us instant updating after a change for free, thus increasing efficiency. The changes are achieved by successively modifying each definition, using regular and general (on lists) arithmetic operations. As we said before, the actual reaching definitions are only stored at the points where needed (only for array references), and once the actual processing - generalized folding - is done, all superfluous information is discarded[10]. For the actual implementation details of this module, see Appendix II.6.

3.2.2.5 Variable folder

In practice, the generalized constant folder and the variable folder are closely related and have to work together. The only difference between them is conceptual, in that the variable folder folds variables (in much the same way as constants are folded), in and beyond basic blocks. Together the two modules produce arbitrary expressions. These may be entirely numeric (possibly containing multiple values - represented as lists), in which case they can be precisely compared. Alternatively these expressions may contain some variables, in which case we can be certain that these are **irreducible** variables. That is

[10]The actual folding is done only on variables which are of direct relevance to the disambiguation process. That is, we start with variables directly involved in index computations and expand backwards as far as possible.

they cannot be compacted and thus must be read as input or result from unpredictable references (otherwise they would have been folded). In any case, the task of comparing two expressions is greatly simplified by the work of the variable/constant folders. As an efficiency issue, the actual information thus obtained is stored only where needed, and any leftover reaching definitions are discarded.

To clarify the above descriptions of variable and generalized constant folding consider the following example:

```
                               Reaching definitions at (*)
    a=b                                 a  ....    (b , b+2)
    c=b+1                               b  ....        ?
    if (x.ne.0)                         c  ....       b+1
       then a=c+1                       d  ....        ?
(*) e=a+b+c                             e  .... (3b+1 , 3b+2)
                                        x  ....        ?
(1) ...ref[e]  ...
(2) ...ref[3b]...
```

Notice how the we have multiple definitions of **a**, reaching statement (*), and thus we obtain a list of values for **e** as well.(**e** is an index for some array, or else none of this information would be computed, and the definitions would have been discarded). Also notice how this makes the disambiguation of references (1) and (2) trivial, when they would have been impossible to disambiguate without variable/constant folding.

Of course, this can easily become computationally expensive to maintain. As usual we rely on propagating pointers to speed-up processing. These pointers are common to all the propagating definitions (but **not** to the original statements so as not to modify the actual code), and thus we again obtain free propagation. The efficiency is further enhanced by the use of data structures which allow relatively fast access to the information needed in this phase. For example the reaching definitions are put into the form: (var def1 def2 ...) where def is either a constant, an expression, an operation or a list. This allows for fast access during the constant/variable folding process. Of course, after constant/variable folding we may be left with such structures which are nested several levels deep[11]. For more details

[11]Since we don't yet make use of conditional information associated with particular paths, this nesting can often be reduced.

see Appendix II.

3.2.2.6 Loop analyzer

The loop analyzer consists of two separate parts. The first attempts to identify loops. The fact that we don't rely on the translator program to supply this information, allows us to attempt to handle more general loops (i.e. than FOR-loops). We do assume however for the success of our algorithms that the program graphs we have to deal with are reducible[12]. Loops are found using the conventional method described in the "Dragon Book" [1]. The idea is to find the backedges in our program graph (i.e. edges whose head dominates their tail). This can be done easily and efficiently on reducible graphs, using depth first search, hence our restriction. This is achieved by observing that an edge a-->b is a backedge iff DFN[a]$\geq$DFN[b]. Once we have found all the backedges it is easy to actually find all the statements in a loop: we find the ***natural loop*** of a backedge a-->b by finding all the nodes which can reach a without passing through b. This will identify loops, but will not separate nested ones. For space efficiency and ease of use, we actually break these loops apart before proceeding. For more details see Appendix II.

Once loops are found we have to prepare them for memory disambiguation proper. The treatment of loops has to be special, since different rules apply inside them. For example a statement such as: "x:=x+1" means very different things in sequential and loop code. (in straight line code it simply means that x's value is increased by 1, whereas in a loop it is a recurrence and the value of x depends on the initial value (before the loop) and the iteration number. The program in its current implementation deals with three (simple) forms of recurrence:

a) $X := X + c$ Where c can be a constant OR an expression formed of constants and/or variables which are loop invariant.

b) $X := Y + c$ Where c, c' are as above and X and Y are mutually
$Y := X + c'$ referencing and thus form indirect recurrences.
This is replaced (not in the actual code) with:

---

[12]This is also assumed in generating **trace-fences** (see next chapter).

$$X = Y_0 + c + (i - 1)*(c' + c)$$
$$Y = Y_0 + i*(c' + c)$$

c) X := Y + c Where c,c' are as above, X is dependent on Y, and
Y := Y + c' Y is an induction variable.

The general form of mutually referencing recurrences is not necessarily much harder to handle but since it involves very expensive computations (e.g. raising a matrix to the n-th power — for iteration # n), and may not turn out to occur too often, was left out for the time being. For a method to deal with this, see the last section of this chapter. Notice that the fact that we don't handle such recurrences simply means that we leave them as separate variables, rather than put them in standard form. While this may involve a loss of accuracy, it allows disambiguation to proceed.

The program has to first locate the recurrences (this is done before the generalized constant/variable folding since as explained above folding on recurrences would give erroneous results). We achieve this by examining the actual statements in the loop body and making a list of the variables each references. For all the variables being defined in the loop body we form an NxN matrix on which we perform a transitive closure, to detect recurrence statements. (Note that we *do handle* chains of variables which end with a recurrence — as in case (c) above; We just don't reduce chains of *mutually* referencing recurrences longer than 2).

When this is done we check each of the recurrences for one of the forms we handle (with a simple minded pattern-matcher). If one is found, we replace the recurrence with its closed form solution *in* the IN reaching definitions of the loop (this is propagated automatically because of the identical pointers used). Note that processing the recurrences involves getting some information about the number of times the loop is executed and about initial values. Initial values information is extracted without any problems if available, by simply detecting reaching information just before the start of the loop. Finally the generalized constant and variable folding is done on the loop body, propagating the new information - about recurrences - to each of their uses. Propagation of information not involving induction variables proceeds as in ordinary sequential code. See Appendix

II.7 and II.6 for more details.

3.2.2.7 Algebraic expression normalizer

This module simplifies symbolic expressions and puts them in a canonical form. This enables the actual comparison mechanism to do a good job. This is incidental to the whole memory disambiguation problem, but the performance of the whole system is dependent on it. However since it is of no conceptual importance for our work, (and so as not to get bogged down in details and never get the system itself off the ground), we have only developed it enough to satisfy our initial needs. As such the module deals with only addition , subtraction and multiplication both on single elements and lists, and handles the usual (trivial) special cases such as: $x*0 = 0$, $x*1 = x$, $x+0 = x$. It also (mainly) puts expressions (in tree form) in a canonical order, so as to facilitate the comparison process attempted by the actual disambiguation mechanism. To do this we use generalized operators (e.g n-ary plus, minus and multiply) [21]. The simplification process is conceptually simple, but very tedious despite Lisp's ability to deal well with tree structures, lists and function application. This is due to the existence of a large number of cases which have to be checked every time: a term is either a number, a list of numbers, a variable, an expression, or a combination list. All the operations defined for constant and generalized constant folding can deal with these new, non standard structures and operators, so that there is no need to convert expression trees back and forth. This allows this module to also interact with the constant and variable folding process. Its output organizes terms - where possible - in the following descending order: variable-lists, variables, lists, constants.

3.2.2.8 Simple range analyzer

This is not really a self contained module but is part of the disambiguator mechanism. It expects to receive (presumably from the program translator) interval bounds, or actual lists of values, (numeric or mixed with irreducible variables). The range analyzer attempts to extract from this data information on whether the values of the variable are either monotonically increasing or decreasing, or the range intervals are overlapping. This module is also responsible for the decision to expand or not to expand variables into lists of values

(or try to solve equations using the variable names) and allows only two variables in diophantine equations.

The information obtained by this module serves to refine the results of the disambiguation, in an attempt to improve accuracy. While its abilities are very limited (just what was mentioned above) these are often encountered occurrences of regular index ranges, and thus could be quite useful in refining the results of the disambiguation proper. Unfortunately, because the program translator was never equipped to provide such information, this module has only been exercised experimentally. (All the other modules have been used extensively as part of the original Bulldog compiler.)

3.2.2.9 Memory Disambiguation proper

This is the final module, which is to use the information produced by all the others to answer the ultimate question: " Do references x and y collide? ". The system tries to answer the above question as referring to two references associated with two particular blocks. The program will differentiate between three cases: first, both blocks are in loop free code, second one or both are in a loop (or different loops) but the question refers to particular iterations, and third given two references inside loops, find if there are any possible collisions at all (for any iteration). The third case really subsumes the other two, but there are slight differences in the information supplied to the program - in a real situation - by the scheduler, which may actually improve the accuracy of the results. In the actual comparison the cases handled by this first version of the program are[13]:

- Both references are constants or constant expressions:
  We just evaluate and compare, obtaining a precise answer.
- Same single variable appears in both references.(e.g. A[ax+c],A[a'x+c']):
  We just solve for the variable, (e.g. x =(c' - c)/(a - a') , and check if the result is an integer).
- Multiple values are involved (e.g. A[i],A[i'] where i in (a1 .. an), i' in (b1 .. bm)):
  We simply check the intersection of these two sets. (for a loop, where these are

[13]Note that all variables encountered are irreducible, at least for the current disambiguator - or else they would have been replaced in the previous passes.

induction variables, we can find the exact iterations between which collisions occur. Note that if large lists are involved we may want to ignore the information (and instead solve diophantine equations) just to avoid an expensive calculation. In our experience so far this has not been necessary, since our translator does not supply array or loops ranges, and those obtained from folding tend to be manageable.

- More than one variable:
  We solve diophantine equation (we currently deal with only the two variables case. This is very likely to be the most common case, particularly for references in loops which are in the same array (or else they couldn't possibly collide). Note that for loops this does NOT mean that references must only contain a total of two variables, but that we will reduce the equation to only two variables, and only solve for these. Thus for example if we had: $2x+2y+ z+1=0$. We would transform it to $2w+z+1=0$ and solve this. While this is trivial to extend to a more general solution, we have not felt the need to do so in the current implementation. For the actual solution to the diophantine equation, see Appendix II.5.

Of course, whenever a precise distinction cannot be obtained the conservative way out is taken and a conflict is assumed[14].

3.2.2.10 Modified live-dead analyzer

Live-dead analysis is not needed for the disambiguation, except to provide correct preemptive conditional jumps dependency edges to the Trace Scheduler. The analysis itself is done trivially, using the formulas:

$$TOP_i = [BOT_i - WRITEREGS_i] \cup READREGS_i$$
$$BOT_i = \mathrm{Union}_{s \text{ in } successors(i)} TOP_s$$

These are applied repeatedly to all instructions until the TOPs all settle down. There are various modifications that we introduced to improve efficiency. Except for the first pass, information is computed on a block by block basis, and only stored at the leaders of each block. (As it turns out that's all we need, since Trace Scheduling needs this information only at the targets of branches. Furthermore, the killed definitions for each block are stored

---

[14]Because the code generator may actually perform certain optimizations if two references are known to refer to the same location, the safe thing to return for unpredictable references is "POSSIBLE-OPERAND-CONFLICT" and not just "OPERAND-CONFLICT" (see interface description in Appendix II.1).

rather than recalculated. These optimizations dramatically improve the speed of the analysis. Note that this will be most efficient when the list is in depth first order (with the block executed first given first).

In order to deal with array references more accurately, (killing the whole array when one element is written is clearly unacceptable for our purpose), we've modified live-dead analysis to actually do some simple checking of array indexes and compare them, using the reaching information already produced. While this works pretty well for our current needs, it is obvious that we could do much better by using the full power of the disambiguator to refines such discriminations. We will later discuss such a scheme in some detail.

3.2.3 An Example

To make the abilities of the disambiguation system clearer, consider the example in figures 3-2,3-3. The system receives the program and performs the initial analysis (reach,loop analysis, constant/variable folding). At that point it is ready to interact with the trace picker and the code generator/scheduler. Suppose a trace was picked in which references (a) and (b) (in fig.3-2) are both present. The relevant reaching information at these two statements (which is all that would be left from the analysis phase at this point) is also shown in the figure. Notice that C has been constant folded, and K has been variable folded, while I and J have been identified as induction variables and transformed to functions of the standard loop variable, i. All of these have also been placed in a canonical format by the simplifier.

In Figure 3-3 we can see how the actual comparison proceeds: the two indexes are equated and the results are placed into canonical form. Notice how the expansion of K results in two diophantine equations, either of which can cause collisions between the references to occur. The equations get solved in the usual way, reaching the conclusion that no collisions can occur, since neither of the two can actually have integer solutions. It is interesting to notice that this conclusion can be reached, even though the values of K0, K1, K and N are not known at compile time.

```
        read(K0);
        read(K1);
        read(N);
        C:=2;
        if (N > 0) then
           K:=  K0 + K0;
        else
           K:= -K1 - K1;

        I:=2;
        C:=C+1;

  loop:if I >= N then goto exit;
           J:=I+1;
           .....
        a: ref(A[I-4*K+C]);     Reaching defs at (a):
                                C: 3
           .....                K: { 2*K0, -2*K1 }
                                I: 2+(i-1)*(1+1) => 2*i
           .....

        b: ref(A[2*J-K]);       Reaching defs at (a):
           I=J+1;               K: { 2*K0, -2*K1 }
                                J: 2+1+(i-1)*(1+1) => 2*i+1
           goto loop;
  exit:    .....
                                Where i is the standard
                                loop index:[1,#iterations]

   Equation to be solved:
     I-4*K+C = 2*J-K
```

Figure 3-2: Disambiguation example

```
=>
   2*i-4*K+3 = 2*(2*i+1)-K
=>

  3*{ 2*K0, -2*K1 } + 2*i = 1

=>

  { 6*K0, -6*K1 } + 2*i = 1

i.e. two diophantine equations:

   6*K0 + 2*i - 1 = 0

  -6*K1 + 2*i - 1 = 0

Neither of which have solutions ( gcd(6,2)=2 and  rem(1,2)#0 )

Thus no conflicts can occur  and (a) and (b) can be scheduled
independently.
```

Figure 3-3: Solution for the Diophantine Equation in Fig.3-2

3.3 Problems with the Dynamic Interaction with Trace Scheduling

Intuitively, several of the modules described in the previous section seem to encounter special problems which arise from the dynamic changes introduced in the process of compaction by the trace scheduler. These problems were a major concern throughout the initial development stages of the Bulldog compiler and led to long but inconclusive discussions. These problems ultimately led to the formalization of Trace Scheduling and the proof of correctness found in the next chapter. Because the problems first arose in the process of building the memory disambiguation mechanism and motivated much of the later work in this thesis, we will identify the the problems at this point. A more complete description of the problems involved, together with the answers to the questions they raise will be given after the formalization of Trace Scheduling and its correctness are shown. 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

disambiguation mechanism may be impaired and methods which should be used to ensure correctness and accuracy. Fortunately, the changes needed are minor and accuracy and efficiency are essentially unaffected.

3.3.1 Reaching Definitions

Reaching definitions are crucial for the success of disambiguation. They are used extensively to refine the index variables of references that are actually compared, as part of constant and variable folding. Since reaching analysis can be relatively expensive in terms of computation time, it is traditionally (i.e. in optimizing compilers which use it at all) done only once at the beginning of the optimization process. In the presence of the trace scheduling transformations however, the notion of definitions reaching a point in the program (e.g. "definitions reaching the beginning of some block") becomes fuzzy. It isn't immediately clear what points, if any, are equivalent before and after compaction. For example, in figure 3-4, it isn't clear which point (if any) in the compacted program corresponds to point p in the original one, for the purpose of reaching definitions.

This led us to the strong suspicion that trace scheduling transformations change the structure of the original program so much as to render static reach analysis meaningless.

3.3.2 Live-dead Analysis

A similar problem, but even more acute than for reach analysis, is encountered in trying to provide preemptive conditional jump dependencies for trace scheduling. Unless the live-dead information used is accurate, operations may be allowed to move above conditional jumps even when they are not supposed to (we'll deal with this more in the following chapters). This will result in illegal code being produced. Computing live-dead information is also expensive and doing it repeatedly may well be prohibitive. On the other hand, renouncing code movements past conditional jumps would consist of a de facto limitation of compaction to basic blocks, which is also unacceptable. Unlike reaching definitions, the points where live-dead information is necessary are well defined both before and after compaction. Unfortunately, once we realized that, it became easy to see that live-dead

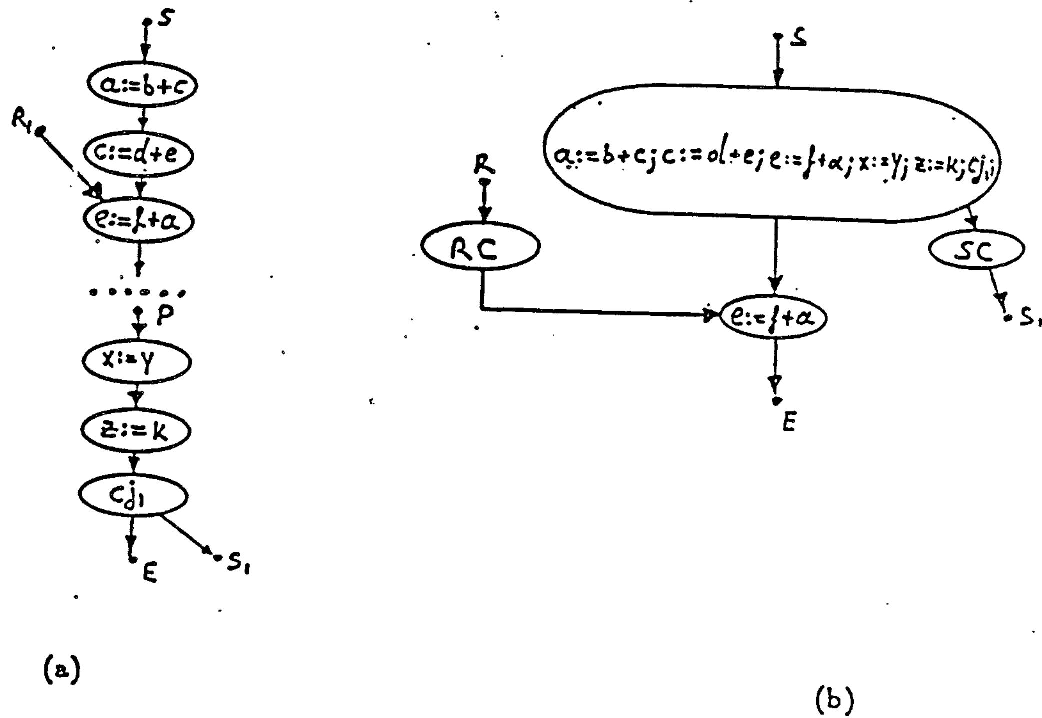


Figure 3-4: Reaching definitions:
Which point in (b) corresponds to p in (a) ?
(a) Before compaction
(b) After compaction

information may change as a result of compaction at those particular points. For an example of this effect see figure 3-5. This further increased our premonition of impending doom.

3.4 Possible Extensions

Other than the solutions to the problems outlined in the previous section, there are a multitude of improvements which are less fundamental, but which have not been implemented. We will outline here several which may be added to improve the performance of future versions of the Memory Disambiguation system.

The most obvious extension to our system is a fancier algebraic analyzer/simplifier.

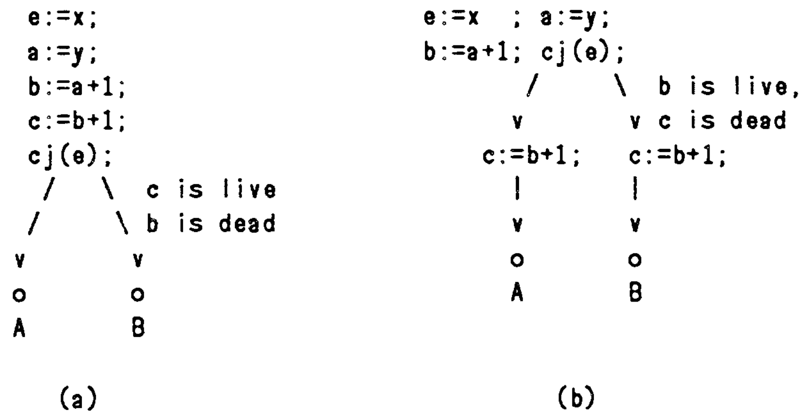


Figure 3-5: Live-dead information changes:
(a) Before compaction
(b) After compaction

As it turns out by examining the results of the compaction, the major obstacle in taking advantage of the full power of the disambiguator is our inability to always simplify expressions properly. The addition of division and possibly logical operators (which will also be needed for some of the following ideas) coupled with better and more complex heuristics for identifying possible simplifications would be a tremendous help. We originally planed to integrate a system like REDUCE [16] into our own programs to deal with this problem, but felt that for our initial implementation we could manage without it.

Another likely extension would consist of the refinement of value-lists for **constant/variable folding**. We could associate the original conditions with values resulting from their respective paths. This would have the merit of making the disambiguation process as tight as possible at compile time. This could be further enhanced by using the **range analysis** techniques described by Harrison [15]. A related but much simpler enhancement which is trivial to implement is the ability to identify and compare odd and even ranges of solutions and induction variables. Most of the mechanism which would make this possible is in place in our system, and despite its special purpose nature, cases where this could be useful may occur often enough as to justify automation[15].

[15]If this is not the case, user supplied information to aid disambiguation, may be more appropriate.

As we have seen, our current system deals with only a limited number of mutually referencing variables in loops. Our **loop analyzer** could be changed to handle more general cases. The general form of mutually referencing recurrences is not necessarily much harder to handle but since it involves very expensive computations (e.g. raising a matrix to the n-th power - for iteration number n), and does not turn out to occur too often, was left out for the time being. To handle it we simply have to rearrange the loop body so that all LHS of assignments to recurrence variables inside one (new) iteration refer only to values from the previous iteration, e.g. all assignments in the nth iteration refer to values computed in the nth - 1 iteration. To get this we simply take the problematic assignments out, and place them in a prolog to the loop unwinding the loop body until we reach these same statements from the (originally) next iteration. After the loop we would have a postlog including the statements that were in the original loop body and are not in the prolog. Then we would decrement the number of times the loop executes by 1.

Finally, we could use a more general solution to (linear) diophantine equations[16] in our **equation solver**. The solution for diophantine equations in more unknowns can be found in any Number Theory book, and is easy to implement. However our handling of loops (using unwinding coupled with trace scheduling) and the discriminations used in building the equations (differentiating among several cases) coupled with our other techniques (variable folding in particular) have proven in our experience to be quite satisfactory in limiting the number of variables we have had to deal with so far.

[16]We only deal with linear diophantine equations. Non-linear ones occur too infrequently (if at all) to justify the effort

Chapter 4

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 become 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. In this chapter we will deal with the formalization and proof of correctness proper. In the next chapter we will answer the questions which truly motivated this exercise.

We will prove that trace scheduling is correct by showing that compacting one trace yields a new program which is indistinguishable from the original one in terms of its input/output behavior. Then we will 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 properties which, if preserved throughout the process, will guarantee the correctness of the transformation. We then proceed by a case by case analysis, to demonstrate that any legal application of trace scheduling transformations will indeed

preserve these properties.

To prove that the process of trace scheduling itself terminates we will show that any legal *composition* of trace scheduling transformations will restrict the ability of the trace scheduler to pick new traces[17]. We are able to show that for any new path of a given length created during trace scheduling, a longer path is removed, by all possible composite transformations. That this is enough to ensure termination is shown more formally by a technique similar to that developed by Manna [28].

4.1 Preliminary Definitions

The following are a series of definitions that are going to be used throughout the description of the model and the correctness proofs:

Definition 1: A **parallel program graph** is a directed graph in which:

- Each node corresponds to one or more operations.
- There is a unique start node.
- Each node is labeled **compacted** or **uncompacted.**
- Nodes labeled uncompacted have outdegree ≤ 2.

Operations which can be in a node are ordinary NADDR operations (e.g. (ADD a b c), (IF-INE i 1 .9 l1 l2), etc.). Uncompacted nodes only contain one operation each.

Definition 2: A program graph is called **sequential** when all its nodes are marked uncompacted.

Initially, all the nodes in a program graph are uncompacted; in the process of compacting the program, some nodes will be transformed and marked compacted. Examples of program graphs can be seen in figures 4-1, 4-2.

Definition 3: A **conditional jump node**, CJ for short, is an uncompacted node with outdegree=2, containing a conditional jump operation.

---

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

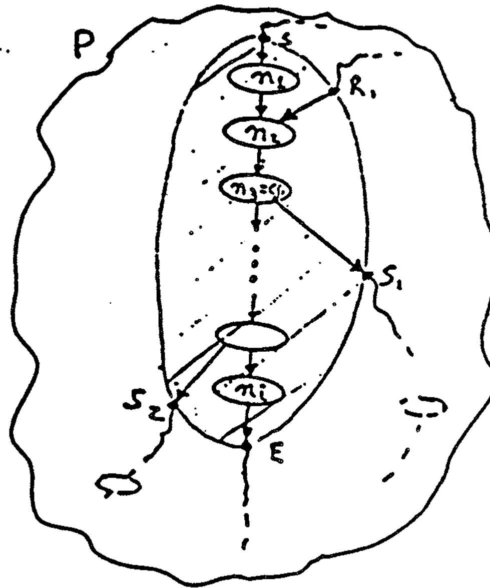


Figure 4-1: A Program Graph

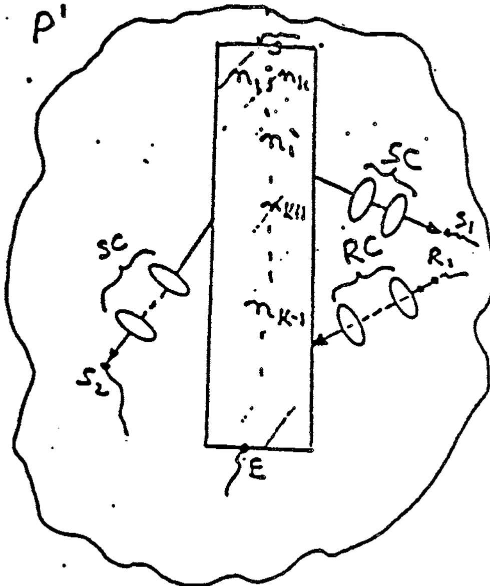


Figure 4-2: A (Partially) Compacted Program Graph

In figure 4-1, nodes with outdegree 2 are CJ's.

Definition 4: A **trace** is a sequence of consecutive uncompacted nodes and intervening directed edges in a (parallel) program graph, containing no repetitions and no backedges.

In figure 4-1, nodes n_1 through n_i, and the intervening edges, form a trace.

Definition 5: A copy of a node n, is another node n', containing the same operations

as n.

In figure 4-2 nodes marked with a ' are copies of nodes with the same name in figure 4-1.

Definition 6: Given a trace in a program graph, we obtain a new program graph containing the original nodes plus a set of pseudo-nodes (no-ops) called **Entry-points** (Start-point, Rejoin-points) and **Exit-points** (Exit-point, Split-point) as follows:

- The **start-point** is inserted on the first incoming edge into the trace (i.e. to the first node in the trace),
- We insert a **Rejoin-point** on every incoming edge into the trace which is not to the first node in the trace, by splitting the edge into two new ones.
- The **Exit-point** is inserted on the last outgoing edge from the trace (i.e. after the last node in the trace), by splitting the edge into two new ones.
- We insert a **Split-point** on every outgoing edge from the trace which is not the last node in the trace, by splitting the edge into two new ones.

In figure 4-1 S, R_i are entry-points, while E, S_i are exit points.

Definition 7: An additional pseudo-node, called a **trace-fence** is introduced across each backedge in the program graph.

See figure 4-1 for an example.

Definition 8: A **trace-path** is a sequence of nodes and intervening edges between an entry point and an exit point in a trace.) The nodes and edges between S and E in figure 4-1 (b) form a trace-path.

Definition 9: A **branch** is a pair (CJ_i,S_i), where CJ_i is an uncompacted node with outdegree 2, and S_i is its associated split-point.

For example, (Cj_1,S_1) is a branch in figure 4-1.

Definition 10: Given a branch, (CJ_i,S_i), the **direction, or cj-dir** of that branch is either T or F.

In figure 4-1, the direction of the branch (CJ_i,S_i) is F.

Definition 11: Given two operations (A,B) and a trace-path or trace in which A precedes B, B is said to be **dependent** on A if:

1. A defines a variable used by B. ($\prec$)

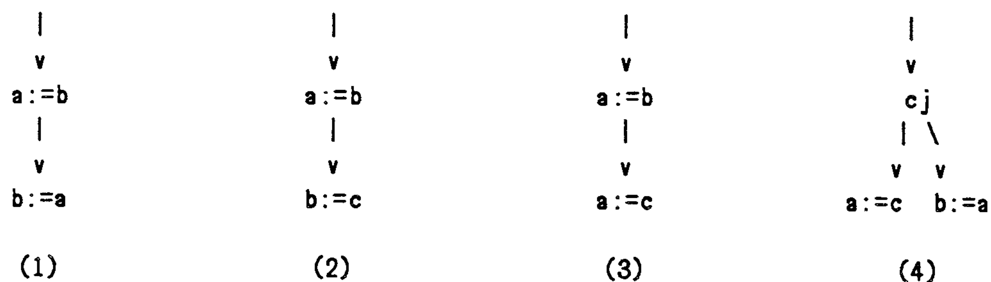


Figure 4-3: Trace Scheduling Dependency Examples

2. A uses a variable defined by B. ($\preceq$)
3. A defines a variable defined by B. ($\prec$)
4. A is a CJ and B defines a variable USED (live) on the off-trace branch of A. ($\prec$)[18]

For examples of all the dependency types, see figure 4-3.

4.2 The Model — A Precise Description of the Trace Scheduler (TS)

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:

4.2.1 Trace Picker (TP):

Chooses (by any method) a trace $t = (n_1,...,n_m)$, such that:

- There exists an edge between n_i, n_{i+1}, where i=1,m-1.
- All nodes in t are marked uncompacted.
- No node in t is a trace-fence.

4.2.2 The Compactor

[18] As we pointed out before, live information seems to change dynamically. For the purposes of the proof this is unimportant (i.e. the way this information is obtained), we merely assume that correct information is always available.

4.2.2.1 Trace Compactor (TC):

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

$N_1=\{n_i | \forall n_j, j<i, n_j \sim\prec n_i\}$

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

$N_k=\{n_i | [(\exists m \in N_k, m \preceq n_i) \vee (\exists m \in N_{k-1}, m \prec n_i)] \wedge [\forall n_j, j<i, n_j \notin \sqcup_{l=1,k} N_l, n_j \sim\prec n_i]\}$

$\equiv$ an operation is placed in N_k if it has either a "$\preceq$ predecessor" in N_k or a "$\prec$ predecessor" in N_{k-1}, and has no dependency predecessors in subsequent N's.

The nodes (N_i's) produced by TC are labeled **compacted**.

(The edges in the new trace t', are defined in section 4.2.2.6).

4.2.2.2 Split Compensation (SC):

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

SC_i = (copy[n_j]|for j:= 1,i-1: ($n_j \notin \sqcup_{l=1,k} N_l$)) $\equiv$ (s_1,...,s_{m_2})

$\equiv$ copies of all operations preceding n_i in t, which don't precede it in t'.

(The edges between these nodes are defined in section 4.2.2.6).

4.2.2.3 Rejoin Compensation - 1 (RC1):

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 is the smallest index satisfying:

(a) j = max{j'|$\forall n_z \in N_i$,$i \geq j$, $z \geq x$, or E (the exit point) if no such j exists}

$\equiv$ 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 = (copy[n_l]|for l:=x,m:($n_l \notin \sqcup_{y=j,m_1} N_y$)) $\equiv$ (r_1,...,r_{m_3})

$\equiv$ 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 in section 4.2.2.6).

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

(c) RC_{cj_a} = (copy[n_z]| for z:=x,l-1:($n_z \notin$(r_1,...,r_{y-1}))) $\equiv$ (q_1,...,q_{m_4})

$\equiv$ copies of operations originally above n_l which aren't in RC_i.

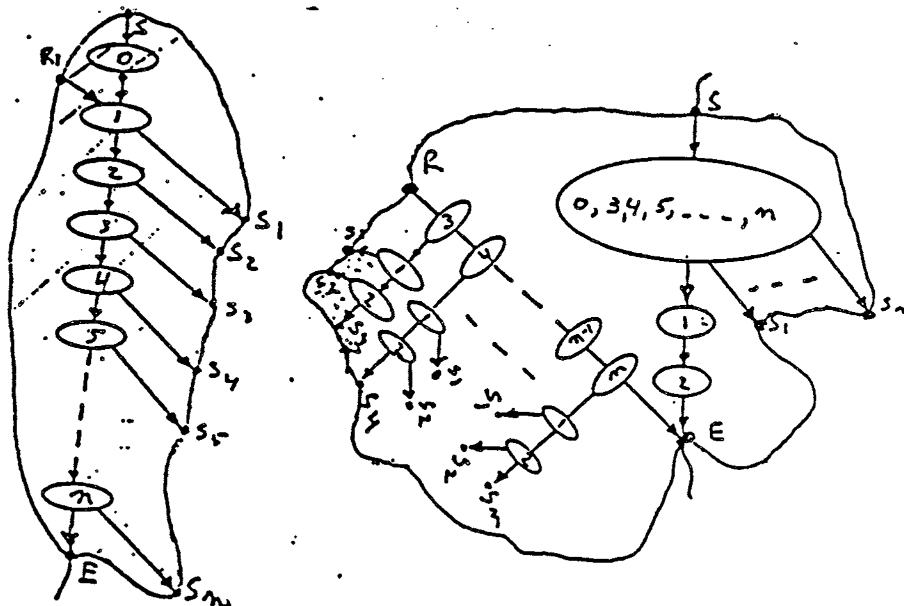


Figure 4-4: Sample of code explosion resulting from RC1

(The edges between these nodes are defined in section 4.2.2.6).

While the above would work, it turns out that it may be extremely space inefficient, and possibly non-terminating. Fortunately, there is a trivial modification which can be made to the above, which will resolve this problem. The space explosion is due to the possible redundant copying of nodes in RC(c) above, which may result in operations being copied several times as part of compensation for different cj's in RC_i. These conditional jumps may create new paths which are actually longer than the ones they replace (see path R->3...n->2->E in figure 4-4 for an illustration of this problem). To remedy this we can do the following:

4.2.2.4 Rejoin Compensation - 2 (RC2):

(a) $j = \max\{j' | \forall n_z \in N_{i,i \geq j'}, z \geq x \wedge (n_z$ is not a cj), or E, if no such j exists$\}$

≡ 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.

Then compensation (b) remains as before, and (c) becomes superfluous. While this is effective in reducing the code explosion in the worst case, and quite likely in general as well, we may do better at the expense of more work:

4.2.2.5 Rejoin Compensation - 3 (RC3):

(a) Given a rejoin to n_z in t, ($n_z \in N_k$ in t'), the ***new rejoin*** in t' is to N_j, where j is identical to that in RC1(a) above.

(b) The rejoin compensation adds a sequence of nodes and edges, RC_i, identical to that of RC1(b) above. Then for $k = \max\{a \mid cj_a \in RC_i\}$, an additional sequence of nodes and edges, RC'_i, is obtained:

(b1) $RC'_i = (\mathrm{copy}[n_l] \mid n_l \in (n_z, \ldots, n_k))$

(b2) $RC_i := RC'_i$

(b3) $j: \forall n_l \in RC_i, (n_l \notin \sqcup_{y=j,m_1} N_y)$

(b4) Repeat b, b1, b2, b3, b4 (for new j) until $RC_i = RC'_i$.

This will avoid lowering the rejoin point by allowing conditional jumps to remain under the rejoin when they can't hurt the compensation. This could be further refined by requiring only cj's to be copied in RC'_i, which would reintroduce the need for compensation type (c) as in RC1 above[19].

4.2.2.6 Continuation Resetting (CR):

For each new node N_i, we define new edges as follows:

Follow(N_i)$=N_{i+1}$; Where N_i is either a node in t', in RC_i, in RC_{cj_a} or in SC_i, and:

- In t':
  $N_{m_1+1} \equiv$ **Exit point.**
- In split copy SC_i:
  $s_{m_2+1} \equiv S_i$, when compensating for cj_i
- In rejoin copy RC_i:
  $r_{m_3+1} \equiv$ **New Rejoin**$_i$, when compensating for R_i.
- In rejoin copy RC_{cj_a}:
  $q_{m_4+1} \equiv S_a$ when compensating for cj_a in RC_i.

The direction of the edges is from N_i to its follower. The continuation resetting affects only the on- trace/trace-path branch of Cj's; the off- trace/trace-path branch stays the same (except of course for the insertion of compensation as described above).

[19]But now RC_{cjk} *would not include any conditional jumps.*

4.2.3 Some Observations

Once we defined trace scheduling properly, there are several observations that can be made and two additional definitions, which turn out to be useful in the process of proving correctness. So as not to detract from the actual proof, we will list them here, for lack of a better place.

Definition 12: A (partially) **compacted graph** is the program graph P' obtained by applying the TS transformation to a program graph P.

An example of a partially compacted graph is given in figure 4-2, together with samples of the transformations described above.

Definition 13: Given P,t and a trace-path p from entrance point S_i to exit point E_j, we call any trace-path e_i from S_i to E_j, in P', an **equivalent trace-path** of p.

For example, e_1 and e_2 in figure 4-2 are equivalent paths for p in figure 4-1.

Definition 14: A path p' in P' is said to be an **equivalent** path of p in P, if p' has the same entrance and exit points as p.

Proposition 1: TS transformations can alter the program graph only between entrance and exit points.

Proof:

- TC only touches the trace body
- SC only inserts copied code between trace body and split points
- RC only inserts copied code between rejoin-points and the trace body or the exit-points.
- CR inserts edges only in the (compacted) trace body or for nodes inserted by the above transformations between entrance and exit points.

All of the above follow trivially from the definition of TS. □

Proposition 2: Paths can only flow from entry points to exit points.

Proof: In the original trace, this is trivially true, by construction (definition) of entry/exit points. In the compacted trace, this is still true by the definition of CR: edges start at entrance points and go towards exit points. □

Proposition 3: A pair (entry-point, exit-point) in the original program graph

uniquely defines a trace-path.

Proof: By definition entry/exit points are unique. If two edges go to or come from the same exit/entry -point, the point is split to guarantee uniqueness. □

Proposition 4: While there may be more than one equivalent path in the compacted graph for a unique trace-path in the original one, all these trace-paths are still between the same entrance points and exit points.

Proof: Only copied CJ's can create new paths in the compacted graph (by definition of TS, no transformation can create new paths, short of copying cj's). Even these paths however will still be between entrance and exit points (because of CR). We'll deal with such paths as they occur. □

4.3 What Exactly Do We Want to Show?

We have now precisely defined trace scheduling and its effect on a program graph. In order to show its "correctness" however, we'll also have to define more precisely what we want to prove. This section attempts to do just that.

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

- Partial correctness
- Termination

where partial correctness and termination are defined as follows:

Definition 15: Trace Scheduling is **partially correct** if for every program graph P and input i, $\mathbf{output(P(i)) \equiv output(TS^{(n)}(P)(i))}$.

Definition 16: Trace scheduling is said to **terminate** if for any input program graph, P, a finite number of applications of TS to P, result in a program graph containing no uncompacted nodes.

Furthermore, to be able to talk about outputs being "identical" we have to define the notion of **execution** in a program graph:

Definition 17: The **execution** of a path through a Program Graph is the evaluation

of operations on it, in the order implicit in the path:

- All operations in $node_i$ are evaluated simultaneously.
- All operations in $node_i$ are evaluated before any operation from $node_{i+1}$.
- Conditions are executed simultaneously, the jump decision is serial. Multiple conditionals in the same node are evaluated like in a lisp Cond statement. See [11] for details.

Definition 18: The **semantics** of a program graph P are said to be preserved by a transformation $T(P) \Rightarrow P'$ if for any input i, executing P yields the same output as executing P'.

4.4 Partial Correctness

4.4.1 Strategy

We will first show that compacting a single trace, t, in P yields P', which preserves the semantics of P. We'll do this by first showing that the relevant parts of the state that need to be preserved to ensure that $output(P(i)) \equiv output(TS^{(1)}(P)(i))$ are:[20]

(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 op(b) is dependent on op(a) in p, and $op(a) \in N_k$, $op(b) \in N_j$, then $k<j$ (if $\prec$) or $k<=j$ (if $\preceq$) in each e_i (≡**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.
   - 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 $op(Cj_i)$), must exist in e_i, with the same direction as the one in p.

---

[20]The conditions defined here are derived from the actual BULLDOG compiler. Somewhat different conditions could also be used.

We then show that (a) and (b) above are indeed preserved by all original trace paths in P and their associated equivalent trace-paths in P' by examining all possible such paths:
I. Start to Exit.

II. Start to Split-point.

III. Rejoin-point to Exit.

VI. Rejoin-point to Split-point.

Of course, in doing so we'll also have to consider all the "byproduct paths" which exist in P' as a result of copied conditional jumps.

Finally we will argue that showing this for one trace is enough, since P' preserves the semantics of the original P, and we can then apply TS to P', and so on. This proof holds for all rejoin compensations (RC1, RC2, RC3).

4.4.2 Partial Correctness Proof

Theorem 1: If for each trace-path p through t, its equivalent trace-path(s) $\{e_1,...e_m\}$ in t' ($\in$P') satisfy:

- Dependency Correctness
- Control Correctness

as defined above, **then** the transformation TS(P)==>P' preserves partial correctness.

Proof:

- A1, A2 imply that all operations in p exist in each e_i, and the dependency order of all such operations is preserved.
- A3 ensures that operations in e_i but not in p don't affect the output at the exit point (from p/e_i), since they can only set unused variables, and cannot affect used variables by violating dependencies with operations from p.

Thus executing e_i will set all variables which are set in executing p to identical values, provided control is maintained on e_i.

- All cj's in p exist in e_i (by a1), their tests results are identical to those in p (by a2 and the above) and so are their off-trace branches and cj-dirs (by b), ensuring that evaluation of cj's in p/e_i result in the same resolutions (i.e. same cj-dirs being taken). Cj's (from p or not) do not affect the output from the

trace-path as long as they maintain control on an equivalent trace-path of p. Otherwise, we are dealing with a different equivalent path e'_i[21], corresponding to an original p', which must also satisfy premises (a) and (b).

□

We have shown that if a and b hold for a trace-path p and its equivalent trace-paths, the outputs produced by executing two paths differing only in p/e_i are identical. Thus if a and b hold for all such trace-paths, the execution of P and P', given identical inputs will produce identical outputs, satisfying our claim.

Theorem 2: Given a trace t in P and the resulting t' in P', [P'= TS(P)],

(a) Dependency Correctness

(b) Control Correctness

will be preserved by all trace-paths through t' in P'.

Proof Outline:

There are 4 basic types of trace-paths in t; we'll show that each type satisfies the premises (a,b) of Theorem 1. For the purposes of our proof we will break down these cases in four separate lemmas, as follows:

I. Start to Exit (S->E).

II. Start to Split-point ($S->S_i$).

We will differentiate between 4 cases:

1. No split compensation for CJ_i is required.
2. Split compensation for CJ_i is required, but does not contain cj's.
3. Split compensation for CJ_i is required and contains cj's.
4. Byproduct trace-paths resulting from cj's in split compensation for CJ_i.

III. Rejoin-Point to Exit ($R_i->E$).

We will differentiate between 4 cases:

1. No rejoin compensation for $rejoin_i$ is required.
2. Rejoin compensation for $rejoin_i$ is required, but does not contain cj's.
3. Rejoin compensation for $rejoin_i$ is required and contains cj's.
4. Byproduct trace-paths resulting from cj's in rejoin compensation for $rejoin_i$

[21]Note that the cj's producing such new paths can only come from the confines of the trace (by the definition of TS transformations) and thus have well defined split-points associated with them.

(cj's in both type b and c rejoin compensation are considered here).

IV. Rejoin-Point to Split-Point (R_i->S_j).

We will differentiate between 5 cases:

1. No rejoin compensation for $rejoin_i$ is required.
2. No split compensation for CJ_j is required.
3. Rejoin compensation and split compensation are required and CJ_j is below the new rejoin.
4. Byproduct trace-paths resulting from cj's in Rejoin Compensation for $rejoin_i$ (subsumes case where CJ_j moves above the new rejoin).
5. Byproduct trace-paths resulting from cj's in split compensation for CJ_i.

Note that these are all the possible trace-paths through P', since "new" paths can be created only as a result of cj's copied as part of rejoin or split compensation, and these are accounted for in II-4, III-4, IV-4 and IV-5. (See also Propositions 1-4 above).

We now examine each of the above types in detail and show how each TS transformation changes it, and show that the resulting path(s) still satisfy the premises of Theorem 1. Our approach is to consider the influence of TS transformations on each type of original path p. When, during compaction, we run into cj's being copied, which create new equivalent paths, we handle them then (i.e. we deal with them when they occur, as opposed to when their original p' occurs). Also note that while this proof specifically deals with RC1, it implicitly applies to RC2 and RC3 as well, since they can be viewed as special cases of RC1, in as much as they differ form RC1 only in that they eliminate the need for type (c) compensation. We will do this by proving the following four lemmas, one for each type of path.

Lemma 1:(Case I) All Start to Exit paths (S->E) in P' preserve dependency and control correctness with respect to their equivalent paths in P.

Proof:

Transformations affecting this trace-path: TC,CR.

- All operations in p exist in e and dependencies among them are preserved (by the definition of TC), thus satisfying a1 and a2.
- Since neither TC or CR add operations not in p to e, a3 is trivially satisfied.
- CR, (by definition), ensures that cj-directions in p are preserved in e, and since

all the cj's in p exist in e, b is satisfied.

□

Lemma 2:(Case II) All Start to Split-Point ($S\text{->}S_i$) paths in P' preserve dependency and control correctness with respect to their equivalent paths in P.

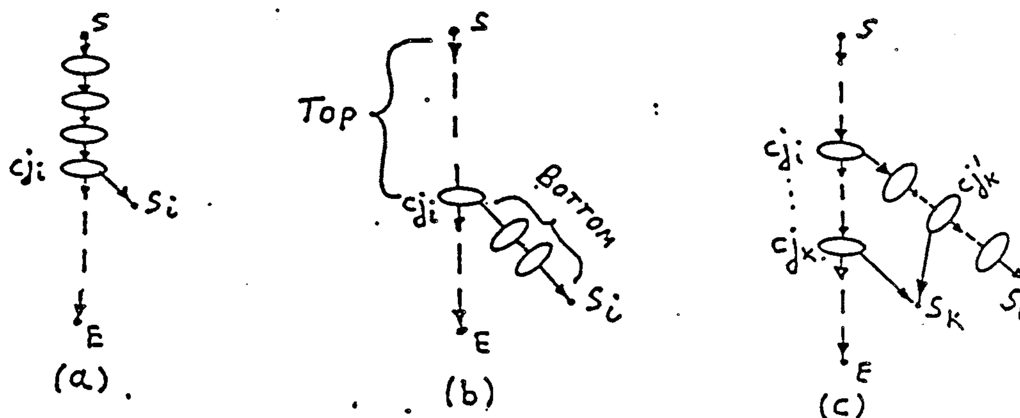


Figure 4-5: Trace-path S->Split-point

Proof: For simplicity, we'll differentiate between 4 cases:

1. No split Compensation [e is on trace up to the last cj, cj_i] (See figure 4-5 (a).)

Transformations affecting this trace-path: TC,CR.

- Essentially a subpath of (I).
- All the operations above cj_i in p are above it in e (since there is no compensation), and no operations can exist between cj_i and S_i - by the definition of S_i. Thus a1 is satisfied.
- Operations above and including cj_i in e, preserve the dependencies existing in p (else there must exist operations in e contradicting the definition of TC), thus a2 is satisfied.
- Operations which may have moved into e (only from below) must pass over cj_i and thus must satisfy dependency#4 (implicit in TC),thus a3 is satisfied.
- All cj's above and including cj_i in p are preserved in e, and cj-dirs are unchanged (by the definition of CR). Any extra cj's introduced in e preserve their directions and dependencies (by CR,TC). Thus b is satisfied.

2. Split Compensation without cj's [e is on trace up to the last cj, cj_i] (See figure 4-5 (b).)

Transformations affecting this trace-path: TC,SC,CR.

- Operations in e divide into two groups: **Operations above (and at)** cj_i

(≡**Top**) and operations in **SC code** (≡**Bottom**). By the definition of SC, all the operations in p are in e (Top U Bottom), satisfying a1.

- Operations in TOP preserve dependencies among themselves (by the definition of TC), and so do operations in BOTTOM (since copied in original order by SC). Furthermore,any op1∈Top and op2∈Bottom can't violate dependencies, since either op1 precedes op2 in p, and then dependencies are preserved in e, or op1 moving above op2, implies op1∼≺ op2 (by the definition of TC). All operations in p existing in e and dependencies among them being preserved,a2 is satisfied.
- A3 is satisfied by TC's enforcement of dependency #4 with respect to CJ_i.
- All cj's on p are in e (actually above cj_i) and they preserve their cj-dirs (by CR), satisfying b.

3. Cj's in Split Compensation[e is on trace up to cj_i] (See figure 4-5 (c).)

Transformations affecting this trace-path: TC,SC,CR.

- Except for cj's in SC code this case is identical to II.2.
- All cj's in p exist in e (all operations in p are in e by the definition of SC), satisfying a1.
- Like all other operations these cj's preserve their dependencies (see II.2), satisfying a2.
- A3 is satisfied ads in II.2 by the enforcement of dependency #4 in TC.
- By CR, copied cj's (like the rest) preserve their cj-dirs, and since all of them exist in e, b is also satisfied.

4. Byproduct Trace-Paths Resulting from Cj's in Split Compensation

Cj_k which is copied as part of SC for cj_i, implicitly defines a 'new' trace-path e', corresponding to the original S->S_k (p'). We must show that this type of trace-path also satisfies the requirements of Claim 1, with respect to p'. (See figure 4-5 (c).)

Transformations affecting this trace-path: TC,SC,CR.

- All the operations above cj_k in p' will be above cj_k in e' (either above cj_i on e', or in SC code above cj_k). Thus a1 is satisfied.
- Dependencies between operations above and including cj_kin e' are preserved (as a subset of the trace-path to cj_i), satisfying a2.
- A3 is satisfied since no operations can move into the off-trace part of e', and any operations moving into the on-trace part of e' must have satisfied dependency #4 *with respect to the original on-trace* cj_k.

- All cj's in p are in e' (like all operations), and by CR they preserve cj-dirs, satisfying b.

□

Lemma 3:(Case III) All Rejoin-point to Exit (R_r->E) paths in P' preserve dependency and control correctness with respect to their equivalent paths in P.

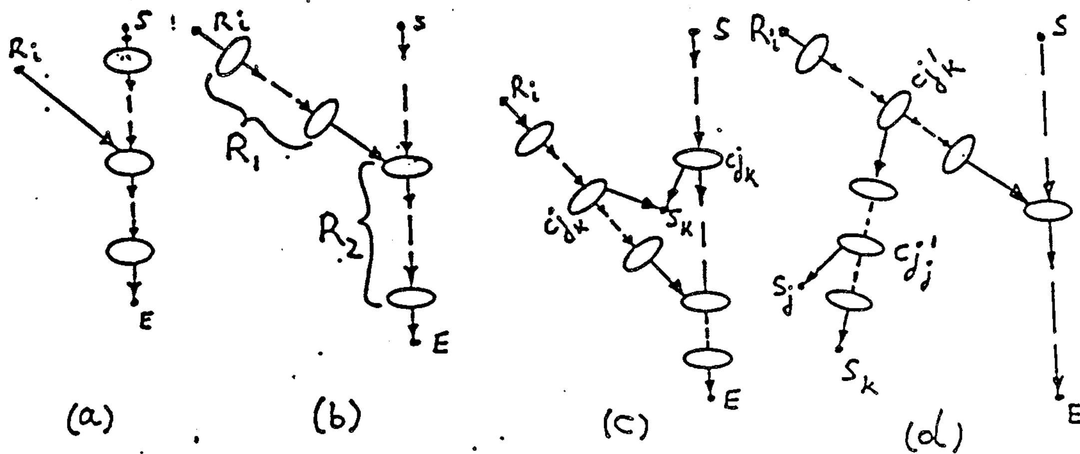


Figure 4-6: Trace-path Rejoin-point->Exit

Proof: Again, for simplicity we will break this in four subcases:

1. No Rejoin Compensation (See figure 4-6 (a).)

Transformations influencing this trace-path: TC, CR, RC(a).

- All operations bellow the rejoin in p stay below it in e, satisfying a1.
- All operations preserve dependencies (else the TC definition would be violated on S->E), thus a2 is satisfied.
- By RC(a), nothing can move into e (from above the rejoin), satisfying a3.
- Cj's in p are preserved in e, and their cj-dirs are preserved (by CR on S->E), satisfying b.

2. No Cj's copied as Rejoin Compensation (See figure 4-6 (b).)

Transformations influencing this trace-path: TC, RC(a,b), CR.

- Operations in e divide into two groups: **Operations in RC(b) code (≡R1) and Operations on-trace below the new rejoin (≡R2).** All operations in p are in e (R1 U R2), by the definition of RC, satisfying a1.
- Operations in R1 preserve dependencies among themselves (RC copies in order) and so do operations in R2 (else we have a TC definition violation). Dependencies are also preserved across the rejoin (same argument as for II.2 but with respect to RC(b)). Thus dependencies between all the operations in p are

preserved in e, satisfying a2.

- No external operations (from above the old rejoin in p) can exist in e (by RC(a,b)), satisfying a3.
- All cj's in p, are below the new rejoin (by assumption) and preserve their original (p) cj-dirs in e (by CR), satisfying b.

3. Cj's Copied as Rejoin Compensation

Transformations influencing this trace-path: TC, RC(a,b), CR. (See figure 4-6 (c).)

- The only difference between this case and III.2 are the copied cj's.
- Like all other operations (see III.2), all cj's present in p will exist in e, satisfying a1.
- Since cj's are also subject to dependencies, they preserve dependencies like other operations (III.2). Thus a2 is satisfied.
- As in III.2, by the definition of RC(a), operations which are not in p, can't exist in e, thus satisfying a3.
- Any cj's missing from bellow the new rejoin are in the RC code (by RC(a,b)), and CR ensures that cj-dirs are preserved, satisfying b.

4. Byproduct trace-paths Resulting From Cj's in Rejoin Compensation

Each copied cj_k results in a new trace-path, e', corresponding to some original R_i->S_k(≡p'). We'll show that such new trace-paths (e') satisfies the prerequisites of theorem 1 with respect to p'. (See figure 4-6 (d).)

Transformations influencing this trace-path: TC, RC(a,b,c), CR.

- All operations in p' are in e' (by RC(b,c,)), satisfying a1.
- Operations in RC(b) and RC(c) code each maintain dependencies internally (original order); dependencies in between RC(b) and RC(c) are maintained (op1∈RC(b) and op2∈RC(c) are either in order or independent by the definitions of RC(b,c) and TC): If op2 is in RC(c) it means that it is below the rejoin in the compacted trace-body; op1 being in RC(b) means it's above the rejoin in the compacted trace-body. So op1, op2 are either in original order, or they are independent of each other.

Thus dependencies among all operations in p are preserved in e, satisfying a2.

- No operations which are not in p are introduced in e' (by RC(a,b,c)), so a3 is satisfied.
- CR preserves cj-dirs, and all cj's in p' are in e', satisfying b.

- Cj's in RC(c) code, say cj_j, create yet other trace-paths,(e"), which also satisfy the requirements of Theorem 1 with respect to their original p", $R_r{\text -}{>}S_j$, since:
  - All operations above cj_j in p" are above it in e" (either in RC(b) or RC(c) code), satisfying a1.
  - Dependencies are preserved (operations in e" are a subset of those in e', thus the same argument as above applies for dependencies across cj_k), satisfying a2.
  - No operations which are not in p" are introduced in e" (by the definition of RC(b,c)), so a3 is satisfied.
  - All cj's in p" exist in e" (since cj's above cj_j in p" are above it in e" - by RC(b,c) on e') and CR preserves cj-dirs, satisfying b.

□

Lemma 4:(Case IV)

All Rejoin-point to Split-Point ($R_r{\text -}{>}S_j$) paths in P' preserve dependency and control correctness with respect to their equivalent paths in P. □

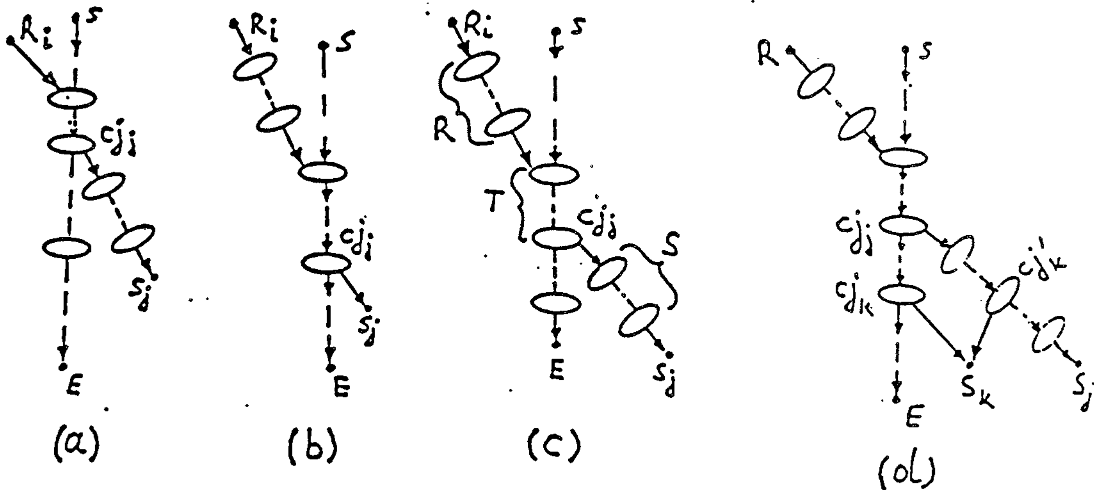


Figure 4-7: Trace-path Rejoin-point->Split-point

Proof: Transformations influencing this trace-path: TC, SC, RC(a,b,c), CR.

1. **No Rejoin Compensation** (See figure 4-7 (a).)

Reduces to case II (the rejoin point acts as the start point for all practical purposes, since nothing moves above the rejoin).

2. No Split Compensation (See figure 4-7 (b).)

Essentially reduces to case III, with the addition that operations from outside p may move into e from below.

- A1 and a2 are satisfied exactly as in III above, not being affected by operations not in p moving into e.
- A3 is also satisfied since operations from outside p have to satisfy dependency #4 with respect to cj_j in TC, in order to enter e.
- All cj's in p are in e and CR preserves cj-dirs, thus satisfying b.

3. RC and SC are required, and Cj_j is Below the New Rejoin (See figure 4-7 (c).)

3 distinct groups of operations exist on this trace-path (e):

- RC(b) Code $\equiv$ R; op1$\in$R.
- On-trace Code $\equiv$ T; op2$\in$T.
- SC Code $\equiv$ S; op3$\in$S.

Note that T$\wedge$S=$\emptyset$, T$\wedge$R=$\emptyset$, by the definition of RC, SC and TC. S$\wedge$R=$\emptyset$ by the fact that Cj_j is below the new rejoin, and the definitions of RC, SC, TC: an operation is in S as a result of cj_j moving upward over it; thus operations in S are below the rejoin on-trace, and therefore by the definition of RC, can't be part of R.

R,S,T each preserve dependencies internally, by the definitions of RC, SC, TC respectively. Dependencies are also preserved across them since:

- If op1 precedes op2 in p => dependencies are preserved in e, and if op2 precedes op1 in p => no dependency between them can exist since op1$\in$RC(b)code => moved above op2 on-trace, during TC.
- If op2 precedes op3 in p => dependencies are preserved in e, and if op3 precedes op2 in p => no dependency between them can exist since op2$\notin$SC code => op2 moved above op3 on-trace, during TC, while op3 is below cj_j.
- If op1 precedes op3 in p => dependencies are preserved in e, and if op3 precedes op1 in p => no dependency between them can exist since op1$\in$RC(b)code => moved above op3, on-trace, during TC, while op3 is below cj_j.
- All operations in p are in e (in R or T or S - without redundancy, as shown). Thus a1 is satisfied.

- Dependencies are preserved among all of them, as shown above, satisfying a2.
- Any operations not in p moving into e, must come from below (by RC(a) they can't come from above) and must satisfy dependency #4 with respect to cj_j in TC, satisfying a3.
- Since CR ensures cj-dirs preservation, (and all cj's are in e), b is also satisfied.

4. Byproduct trace-paths resulting from Cj's in RC Code

(Note that this subsumes the case where CJ_j itself appears as part of RC code).

This case reduces to one of the versions of III.4 or III.3.

5. Byproduct trace-paths resulting from Cj's in SC Code

(Assuming that CJ_j is below the new rejoin in e; Otherwise this case reduces to IV.4 above). (See figure 4-7 (d).)

Subsumed by IV.3, with II-4 for a3. For each CJ_k in SC code creating a new equivalent trace-path, e', corresponding to an original p', R_i->S_k, we have:

- All operations above CJ_k in p' are above it in e' (either in SC code or on-trace or in RC code - by the definition of RC, SC, and same argument as for IV.3), satisfying a1.
- All dependencies among the operations are preserved, (same argument as for IV-3), satisfying a2.
- Operations from outside p' can move into e' only from below (by the definition of RC), and only in the on-trace part of e'. Thus they must preserve dependency #4 with respect to the original on-trace CJ_k (as in II.4), satisfying a3.
- All cj's in p' exist in e' and their cj-dirs being preserved by CR, b is also satisfied.

□

Having proved lemmas 1-4, we have in effect shown that all possible paths through t' satisfy dependency correctness and control correctness as required by theorem 2 and thus the conditions for the applicability of theorem 1 are satisfied. That is, we have shown that one application of trace scheduling preserves partial correctness. We will now show that this is enough to ensure partial correctness after any number of trace compactions.

Theorem 3: Showing that partial correctness is preserved for the compaction of one

trace is enough to show that partial correctness is preserved after n applications of TS.

Proof:

By trivial induction:

Base Case:

Given a trace t in P, the premises of theorem 1 are satisfied for the equivalent paths of all possible trace-paths. It follows that one application of TS to a trace in a program graph, preserves partial correctness. That is, output(P(i)) $\equiv$ output($TS^{(1)}$(P)(i))

Inductive assumption:

Assume this holds for n-1, applications of TS, i.e. output(P(i)) $\equiv$ output(P'(i)), where $TS^{(n-1)}(P) \Rightarrow P'$.

Since the resulting P' is itself a program graph, applying TS to it would yield a new program graph P" which by theorem 1 preserves the semantics of P'. But P' (by the above assumption) preserves the semantics of the original P, thus P" also preserves the semantics of P. That is, output(P(i)) $\equiv$ output($TS^{(n)}$(P)(i)). Thus applying TS until no more uncompacted code is found (assuming termination) will preserve partial correctness. □

4.5 Termination of Trace Scheduling

As we have seen in the previous section, the transformations that trace scheduling applies to a program graph are partially correct, in the sense that they preserve the semantics of the original program. In this section we are going to prove that Trace Scheduling actually terminates. This is not strictly speaking necessary for the building of a real life trace scheduling compiler. In practice we are likely to turn compaction off after the more important traces have been compacted. However, it is still aesthetically pleasing and interesting from a theoretical point of view. Furthermore, the experimental implementation

of the BULLDOG compiler always tries to achieve as large a speed-up factor as possible, so it always applies Trace Scheduling to each trace it encounters. Thus it is important, particularly when working on a large program, to know whether we can expect the process to terminate or whether we should force it to, possibly at the expense of obtaining smaller speed-ups.

4.5.1 Preliminaries

4.5.1.1 Observations

Very informally, non-termination can occur in essentially two ways:

1. Production of traces of increasing length. That is, compensation code being produced[22] results in traces which are longer than the one being compacted. This is impossible, by the definition of the transformations in section 4.2, which never copy more operations *on any given path*, than there were originally in the trace.
2. An infinite number of new possible traces are produced as a result of compaction. That is, the compensation code being produced[5] results in a growing number of paths, which each can then be picked as a trace, and so on. This is by far the hardest case to disprove, since it is easy to see how the compaction of one trace may lead to the creation of several others.

We should also realize that the Trace Scheduling transformations cannot occur in separation. For example, split or rejoin compensation can only occur in conjunction with trace compaction. Therefore it is important to only consider transformations in the context in which they occur, i.e. take them as groups rather than individually.

These are the major issues involved. Keeping them in mind should facilitate the understanding of the proof proper.

---

[22] We are only interested in compensation code, since the nodes in the compacted body of the trace are marked compacted, and thus eliminated from further consideration.

4.5.1.2 Definitions

All the following definitions assume we are dealing with a program DAG (i.e. the program graph has no cycles). We will later show that this is all we need to deal with, in order to prove termination.

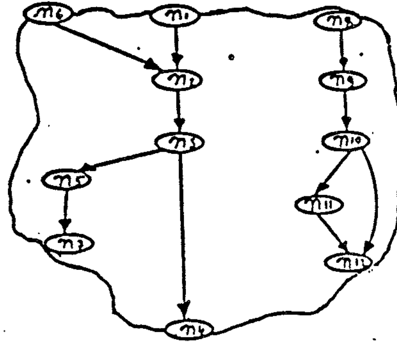


Figure 4-8: Definitions Examples

Definition 19: A **root** of a DAG is a node with in-degree equal to zero.

Definition 20: A **sink** of a DAG is a node with out-degree equal to zero.

In figure 4-8, n_1, n_6 and n_8 are roots, while n_4, n_7 n_{12} are sinks.

Definition 21: A (directed) **path** is a sequence of consecutive nodes and intervening directed edges in the program DAG, from a root to a sink node. We may denote such paths by $Root_i$->$Sink_j$, if only one such path exists between $Root_i$ and $Sink_j$, and thus no ambiguities can arise.

For example, in figure 4-8, nodes n_1, n_2, n_3, n_4, together with the intervening edges form a path from (root) n_1 to (sink) n_4. Notice that we allow program DAG's to have multiple roots and sinks. Also notice that there may be multiple paths going through the same nodes and edges (of course if they totally overlap they are the same). For example, The path from n_6 to n_7 shares nodes and edges (all except for the roots and sinks and the respective edges) with the above path n_1->n_4.

Definition 22: A (directed) **subpath** is a sequence of consecutive nodes and intervening directed edges in the program DAG, from some node to a another.

Definition 23: The **length** of a path is the number of nodes on that path.

For example, the path formed by n_1, n_2, n_3, n_4, and the intervening edge in figure 4-8 has length four, the one from n_6 to n_7 and the intervening edges has length 5.

Definition 24: The **height** of a DAG G is the length of the longest path in G.

Definition 25: The **height of a node (or entry/exit point) n** in a DAG is the length of the longest subpath from n to a sink node.

The height of the DAG in figure 4-8 is 5, that of node n_3 is 3. Note that there may be more than one longest possible path or subpath.

4.5.2 Proof of the Termination of Trace Scheduling

4.5.2.1 Outline

We want to show that during compaction the height of the DAG continuously (if slowly) decreases. To do this we will examine all possible ways in which Trace Scheduling may change the program DAG during the compaction of one trace, and try to prove that:

1. At least one path decreases in length, as a result of the compaction and removal from further consideration of the trace-body.
2. All split compensations result in *new* paths which are at least 1 node less high than before compaction (thus ensuring that eventually the height of the whole DAG decreases).
3. All rejoin compensations result in paths which are just duplications of paths which are eliminated as a result of the removal of the trace body from further consideration, and thus can have no negative (or positive) effect on termination.

If we can show the above, we will be done (at least for the DAG) since each TS compaction step either introduces new shorter paths and reduces some subgraph height, or Removes a rejoin and reduces some subgraph height, and none increases the overall height. Thus the height of the DAG will ultimately decrease, and the process will terminate.

To make the above into a rigorous argument we will show that:

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 length***, that is, l_k precedes l_i iff $k>i$. Furthermore, a lexicographic ordering is defined on the tuples.

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

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

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

This proof applies to the modified rejoin compensations (RC2, RC3). RC1 may create longer paths than those it removes, and also increase rejoin height. While this does not necessarily result in non-termination, it does indicate that worst case behavior may be much worse than that of the modified compensations. More experimental results are needed in order to establish the best approach in a practical system (which may switch trace scheduling off after a fixed number of traces).

4.5.2.2 Termination Proof proper

We will now consider all the possible transformation groups and show that they all have the format of equation 1.

First, we'll show that there are only four possible groups of transformations which may occur during compaction.

Theorem 4: The following are the **only** transformation groups which may occur during compaction:

1. Trace Compaction
2. Trace Compaction and Split Compensation
3. Trace Compaction and Rejoin Compensation

4. Trace Compaction, Split Compensation and Rejoin Compensation

Each of the above also include Continuation Resetting, which occurs whenever another transformation occurs.

Proof: We will only deal with the three basic transformations: TC, SC and RC. The fourth transformation (Continuation Resetting) is only a byproduct of the others and only occurs when one or more of the others occur. There are eight possible combinations of the three possible transformations[23]:

1. TC, (no SC), (no RC)	(1 0 0)
2. TC, SC, no RC	(1 1 0)
3. TC, no SC, RC	(1 0 1)
4. TC, SC, RC	(1 1 1)
5. No TC, SC, RC	(0 1 1)
6. No TC, SC, RC	(0 1 0)
7. No TC, SC, RC	(0 1 1)
8. No TC, SC, RC	(0 0 0)

We can notice that cases 5-8 above can never occur, since by the definition of trace scheduling, split and rejoin compensations can occur only as a result of compaction of a trace, (i.e. TC). Furthermore case 8 above is also impossible since again by the definition of TS, TC must occur. Thus only cases 1 through 4 can possibly occur during compaction, vindicating our claim. □

Theorem 5: The application of Trace Compaction alone (case 1) to a trace t, results in the decrease of the length of the longest path through t, of length max_t, and the increase of the length of none, while possibly creating shorter paths. More precisely, the application of this transformation group results in:

$$<l_{max},\ldots,l_{max_t},\ldots,l_1> \Rightarrow <l_{max},\ldots,l_{max_t}^{-},l_{max_t-1}^{+},\ldots,l_1^{+}>$$

Proof: The nodes produced by TC are marked "compacted" and the body of the trace

[23]The bizarre ordering of the cases is that in which we wish to consider them in our subsequent proofs, to improve the quality of the exposition.

t (the original nodes being compacted) are taken out of the DAG and replaced with the newly compacted nodes (by the definition of TC). As a result, the for the purpose of Trace Scheduling which can only new DAG effectively contains fewer nodes available for future compactions, by the definition of the trace picker (TP). In the process, one or more "new" paths may be created, since from the perspective of the TP, new roots and sinks appear in the DAG, at the exit and entrance points of the trace, respectively (see figure 4-9). In any case, these "new" paths are the result of cutting all paths through t into pieces while removing some nodes, from their original length (by removing the body of t). Thus any path which is now k nodes long, and is a "new" path, must have been longer before compaction (must have shared at least 1 node with t, or else it couldn't go through t or into t). In particular, the longest path through t, say of length max_t, decreases in length.

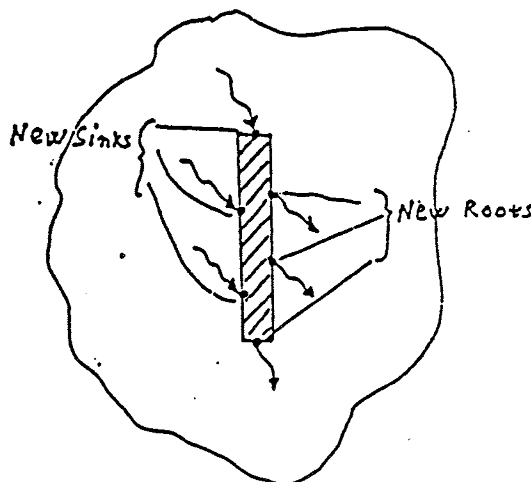


Figure 4-9: Effect of TC on DAG

Thus if originally our n-tuple was:

$$<l_{max},\ldots,l_{max_t},\ldots,l_1>$$

The application of this transformation group to the DAG will result in:

$$<l_{max},\ldots,l_{max_t}^{-},l_{max_t-1}^{+},\ldots,l_1^{+}>$$

That is, the number of paths of length shorter than max_t may well increase by any

finite amount, (since there are only a finite number of paths to start with) but the number of paths of length max_t decreases, and the number of paths of length larger than max_t is unchanged (since max_t is the longest path through t). □

Proposition 5: First node in a trace t, can never be copied as part of any split compensation.

Proof: The first node, n_1 in the trace has no predecessors in t, and thus cannot depend on any other operation in the trace body. By the definition of TC, it follows that it will be scheduled in N_1, and therefore no cj operation in t can be scheduled before it. By the definition of SC, it then follows that n_1 cannot be copied as part of any split compensation. □

Theorem 6: The application of Trace Compaction and Split Compensation (case 2) to a trace t, results in the decrease of the length of the longest path through t, of length max_t, and the increase of the length of none, while possibly creating shorter paths. More precisely, the application of this transformation group results in:

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

Proof: By the definition of Split Compensation only operations which were originally above cj_i and are, as a result of compaction, below it will be part of SC_i. That is, (using proposition 5 above), $SC_i \subseteq \{n_2,...,n_{i-1}\}$ (see figure 4-10 (a) and (b) for an illustration). Since by the definition of TC, trace bodies are effectively taken out of the DAG for the purpose of trace scheduling (since marked compacted), every split from the trace will have created a new root in the DAG, say S'_i for cj_i (see figure 4-10). Furthermore because of the possible copying of cj's as part of SC_i, new paths may be created as a result of split compensation, which didn't exist before, since before SC, no path existed from S'_i through S_k, and after compaction there may be one (if cj_k is part of SC_i); see figure 4-10 for an example. However, any new paths introduced in the DAG in this manner are of lesser length than the longest path removed by TC. That is, the height of the start-point of t is greater than that of any new root introduced by

SC. To see this more formally we'll show that:

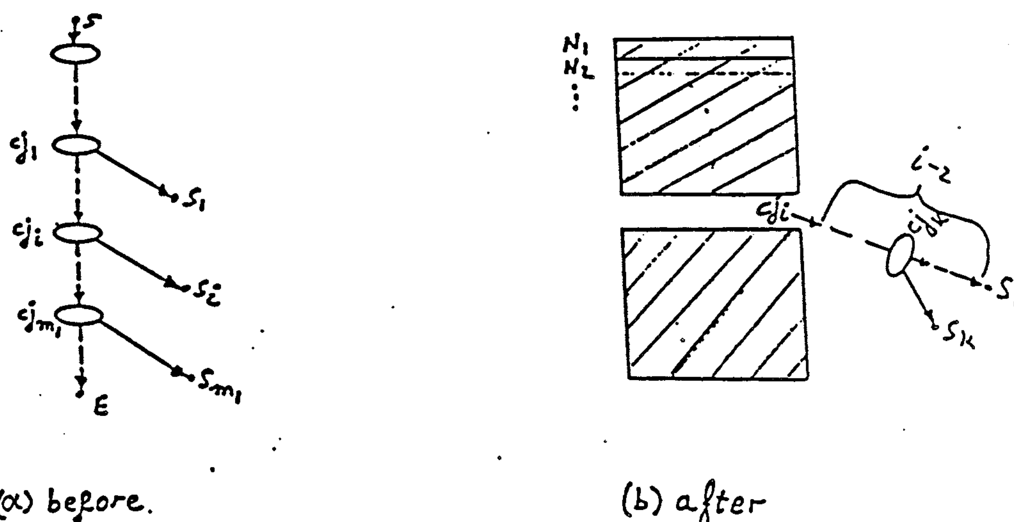


Figure 4-10: Effect of SC and TC on DAG

Lemma 5: $\exists$path p' through t such that $\forall i \in \{1,...,m_1\}$,$\forall$path p from S'_i to an exit point, length(p) $<$ length(p'), where m_1 is the number of conditional jumps in t (see figure 4-10).

Proof: Suppose this is false; then the maximal height of some S'_i must be greater than the length of *every* path through t.

If the longest path from S'_i is through S_i, then by proposition 5 above, the length of the subpath S'_i->$S_i \leq$ i-2 $<$ Original length of the subpath Start-point$_i$->S_i (see figure 4-10 for an example). Thus any path going through Start-point$_i$ and through S_i, is at least 2 nodes longer than any path from S'_i through S_i. Alternatively, the longest path from S'_i may go through S_k, for some copied cj'_k in SC_i. Form S'_i to cj'_k we may only have (including cj'_k) $\leq$ K-1 $<$ i nodes, since at least n_1 will not appear in SC_i (by proposition 5), and cj_k precedes cj_i in t (or else it wouldn't appear in SC_i), whereas the length of the subpath Start-point$_i$->S_k is k nodes. Thus any path going through t and through S_k will have been at least 1 node longer than any path from S'_i through S_k.

Since the above are the only possible paths from S'_i (either through S_i or through S_k , k<i), and both possibilities only create paths that are shorter than the longest possible path through t, it follows that the original assumption is wrong, and thus the maximal height of all S'_i must be less than the length of the longest path in t. □

Thus, by the above lemma, all new paths introduced by SC (by copying cj's) are shorter than the longest path through t. Therefore, when the trace body is removed (as a result of compaction by TC) the longest path p' through t is "segmented", and at least one node is removed from it (same as in theorem 5). Assuming that the length of this longest path in t is max_t, we get at worst, as a result of TC and SC being applied to t, the removal of the maximal length trace through t, and the creation of a finite number of other shorter paths. That is:

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

□

Lemma 6: At most m-1 nodes can be included in any RC compensation type b.

Proof: Rejoin compensation of type b for rejoin i, RC_i, is between the Rejoin-point, R_i and the new rejoin, say R'_i, by the definition of RC (a) and (b) in section 4.2. If $R'_i \equiv E$, then the length of $R_i \text{-} > R'_i$ is = m-x+1, where n_x is the node at which the original rejoin occurred, by the definition of RC (see figure 4-11 for illustrations). If R'_i is to N_j above E, then the length of $R_i \text{-} > R'_i$ is <m-x+1 (by the definition of RC(a,b)).

Thus in either case, $\text{length}(R_i \text{-} > R'_i) \leq$ m-x+1. Furthermore, x>1, since if the original rejoin is to n_1, no rejoin compensation is required (by the definition of RC). Therefore $\text{length}(R_i \text{-} > R'_i) \leq$ m-1. □

Theorem 7: The application of Trace Compaction and Rejoin Compensation (case 3) to a trace t of length m, results in the decrease of the length of the longest path through t, of length max_t, and the increase of the length of none, while possibly

creating shorter paths. More precisely, the application of this transformation group results in:

$$<l_{max},\ldots,l_{max_t},\ldots,l_1> \Rightarrow <l_{max},\ldots,l_{max_t}^{-},l_{max_t-1}^{+},\ldots,l_1^{+}>$$

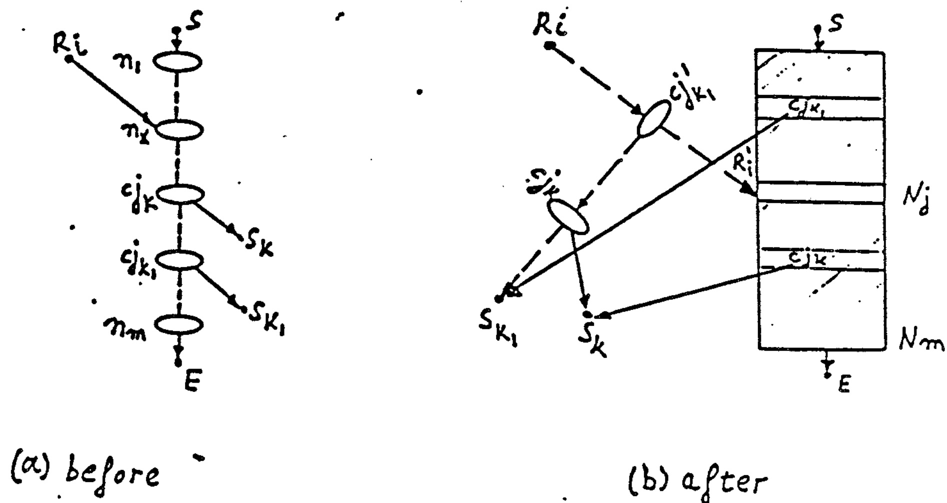


Figure 4-11: Effect of TC and RC on DAG

Proof: Any new paths from a rejoin point R_i are no longer than paths which existed before compaction and *they are actually shorter whenever* $R'_i \neq E$. As we'll show below, this distinction is crucial.

If the new rejoin is to E, all the operations in t below the original rejoin are in RC_i, in original order, thus exactly the same exit points that were reachable from R_i before compaction are reachable after it, through identical paths, of the same length as before compaction, since R_i sees below itself the exact same subgraph as before[24]. The net effect is that no new paths are introduced and none are removed as a result of TC and RC_i, when $R'_i \equiv E$.

If the new-rejoin is higher than E, R'_i is effectively a new sink point, since the trace body is removed for the purpose of future trace picking. Also, any path from a root r

[24] In fact the new paths result from separating 2 subpaths which shared some nodes in t: The one from the top of t through E, and the one from R_i to E.

through R_i to R'_i is shorter than the original path from r through R_i through E, since otherwise the rejoin would be to E (by definition of the new rejoin). Furthermore, paths created as a result of conditional jumps being part of RC_i, i.e. paths through R_i through cj_k, through S_k, are identical to those that existed before compaction, since by the definition of RC2 and RC3, all operations preceding cj_k below n_z (and only those operations) will precede it in RC_i.[25]

To summarize, rejoin compensation (RC2, RC3) by itself will either decrease the length of some paths and live others unchanged (by removing old ones and introducing new but identical ones), or just leave every path from the rejoin point unchanged. Thus, in conjunction with TC, which reduces the length of at least one path, of original length v (while possibly creating more shorter paths as a result of removing the trace body):

$$<l_{max},\ldots,l_v,\ldots l_1> \Rightarrow <l_{max},\ldots,l_{max_t}^-,l_{max_t-1}^+,\ldots,l_1^+>$$

□

Theorem 8: The application of Trace Compaction, Rejoin Compensation and Split Compensation (case 4) to a trace t of length m, results in the decrease of the length of the longest path through t, of length max_t, and the increase of the length of none, while possibly creating shorter paths. More precisely, the application of this transformation group results in:

$$<l_{max},\ldots,l_{max_t},\ldots,l_1> \Rightarrow <l_{max},\ldots,l_{max_t}^-,l_{max_t-1}^+,\ldots,l_1^+>$$

Proof: By the definition of Continuation Resetting, the last (copied) node in either RC or SC or TC has an edge between itself and the original exit point associated with that path (Entry-point->Exit-point). For all nodes (other than the last one) in any of the compensations, edges are defined by CR as going to their successor *in the respective set of nodes.* Furthermore, copied cj's preserve their original off-trace edges. Thus we can never create edges which go between two separate sets of nodes belonging to

---

[25]Notice that this may not hold for RC1, which allows copying of cj's in RC_{cjk}, *which may create new, longer paths since the cj's between R_i and S_k may not be in their original order.*

separate compensations. Therefore the effect of SC, RC, and TC occurring together cannot create any interaction between them (i.e. no new paths may result which go between two different compensations, such as RC_i and SC_j). In particular, this implies that no new subpath Entry-point$_i$->Exit-point can exist in the new DAG if no such path existed in the original one. Thus the net effect of applying SC, RC, TC to a trace t through which the longest path is of length max_t-1, is the cumulative effect of applying each transformation separately:

SC can only create new paths of shorter length than those going through t, while RC can at most remove some paths, while replacing them with shorter ones, and introducing no other new paths. TC is actually going to shorten the longest path through t, while possibly creating new shorter ones (see theorems 5,6 and 7 for a more detailed explanation of why this is so). Therefore we'll still have, overall:

$$<l_{max},\ldots,l_{max_t},\ldots,l_1> \Rightarrow <l_{max},\ldots,l_{max_t}{}^-,l_{max_t-1}{}^+,\ldots,l_1{}^+>$$

□

Finally we'll show that we don't need to concern ourselves with anything but DAG's.

Theorem 9: Trace-fences are enough to enable us to consider only DAG's in showing the termination of trace-scheduling.

Proof: Since traces can't be picked across trace-fences, we can consider a trace-fence as a special start/end of path and proceed as above. That is we can view a trace-fence as defining an entry-point and an exit point into the graph. This will result in possibly multiple start points, which will result in a forest, rather than a tree when building equivalence trees. This is no special problem though as we only care to see that the height decreases (and we have to deal with forests after the first trace compaction anyway). □

82

4.6 Windfall Profits Resulting form the Above

As a result of the better understanding of the Trace Scheduler gained in 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 raised and solved in this process. Furthermore the following potentially important improvements have suggested themselves as a result of the insight gained:

- A conceptual and practical deficiency in the Rejoin Compensation mechanism has been identified, explained and a solution was provided (Rejoin compensation (c) is a result of this process).
- Possibly more effective compensation transformations have been identified: Only copy as compensation operations which define variables which are used (live) below split-points (for splits) or rejoin points (for rejoins). This would of course require semi-dynamic live-dead. 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.
- 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.
- 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). Furthermore, conventional handling of array references in flow analysis methods is more conservative than necessary. In the following section, better methods are described.

While the last item above is not a "windfall profit" since it was actually one of the main motives which caused us to undertake this proof, it follows so easily from the the insight gained in the process of building the proofs that it may well be viewed as such.

Chapter 5

FLOW ANALYSIS, DISAMBIGUATION AND TRACE SCHEDULING

For the process of Trace Scheduling to preserve correctness certain types of flow information (e.g. live-dead) are required. We have also shown that for the BULLDOG compiler to achieve significant 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. incrementaly updated). 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 *local* 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 ***continuous*** dynamic analysis after and during each trace compaction — is totally useless. This would raise serious concerns about the practical applicability of a trace scheduling based compiler in general, and one using extensive memory anti-aliasing in particular, since dynamic flow analysis may well be prohibitively expensive.

Before our formalization of trace scheduling, answering any questions about the

requirements for flow analysis has proven time and time again to be extremely elusive. Intuitively appealing answers where often found, only to be contradicted later on by extremely non-intuitive counter-examples. Now, using the model and proofs of the previous chapter as a basis, we are in a much better position for solving the flow analysis problems which are essential to the success of the BULLDOG compiler.

In the following discussion we will show that static flow information can safely (correctly) be used for disambiguation and trace scheduling purposes. We will also show 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 will 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. In particular, we will show that:

- Static reach analysis is enough for anti-aliasing.
- Live-Dead information at the Entrance/Exit points of a trace doesn't change during the trace compaction.
- Totally static live-dead information is not enough for trace scheduling.
- No update of live-dead information is needed during the trace compaction.
- Incremental and local updating of live-dead information is both correct and accurate for the purposes of trace scheduling. We will also explain how such information can be computed.
- Ordinary live-dead analysis in the presence of array references is not sufficient for the purposes of trace scheduling, and we will outline a more satisfactory algorithm.

5.1 Reach Analysis for Trace Scheduling

It would appear at first sight that since the order of statements can change so dramatically in the process of compaction, no use can be made of static reach information for the purposes of disambiguation. (Recall that the reaching information is used to simplify as much as possible the expressions used as array indexes, so as to allow a more

precise comparison). However, this is not so. The information we need is contained in the static reaching definitions, despite their apparent change. To see this we have to realize that:

1. What we are really after are the *particular definitions* reaching *a particular use* of a variable.
2. It is really meaningless to compare reaching definitions in their traditional sense, at a point p in the compacted program and p' in the original program, since there may not be any way of establishing "correspondent" points in the two programs.

Theorem 10:

For each operation using variable x - use(x) - in P which is reached by the set of operations defining x - DEFS(x), and for each $use'_i(x)$ resulting from the compaction of t in P' and the resulting copying, there exists a set of reaching definitions of x, $DEFS'_i(x)$, such that $DEFS(x) \equiv \bigcup_{i=1,n} DEFS'_i(x)$.

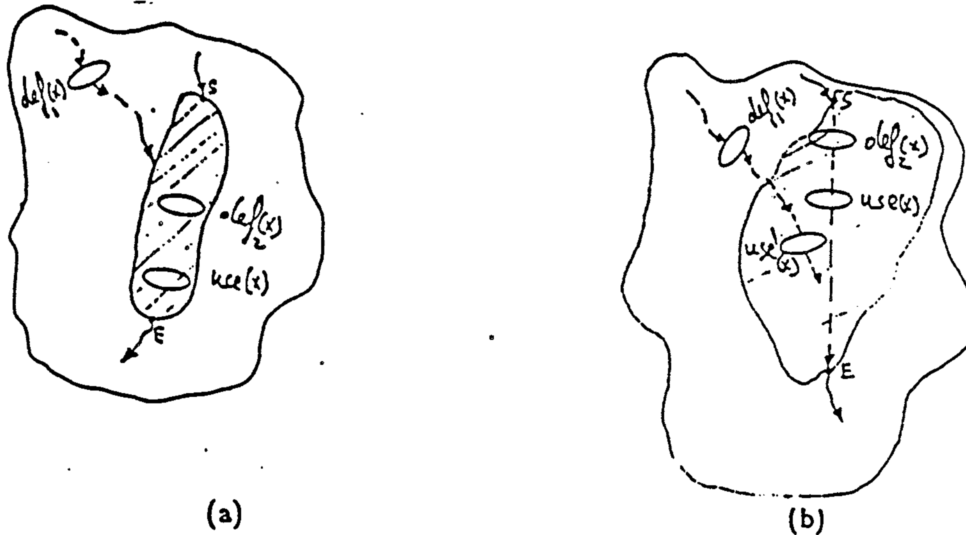


Figure 5-1: Reaching definitions:
(a) Before Compaction
(b) After Compaction

Proof: (Refer to figure 5-1)

(a) $U_{i=1,n}DEFS'_i(x) \subseteq DEFS(x)$

Suppose that the above is false, because there exists a def(x)$\in U_{i=1,n}DEFS'_i(x)$ which does not exist in DEFS(x). However we have shown previously that TS is correct, and in particular that the transformations applied preserve data and control correctness. Thus a definition can only exist above a use (so that it reaches the use) if it existed there on some path in the original program:

All operations in a trace-path p are preserved by its equivalent paths e_i, and dependency order (and therefore definition-use chains) are preserved. Thus no definition of some variable x, def(x), can reach any *instance* of a use, use'$_i$(x) in the transformed program graph, P', without having reached the original use(x) in the initial program graph, P. This is so since:

- Def(x) reaching some use'$_i$(x) in P' cannot be *inside* the code modified by the trace scheduling transformations, since by the correctness proof in the previous chapter, all operations in any p are in all its equivalent e_i's, dependencies between them are preserved, and any operation (and in particular def(x)) moving into a path can't violate any dependencies. In particular dependency #4 is thus preserved preventing an operation to move into a path (above use'$_i$(x)) if it writes a variable which is live on that path. Nor can def(x) have entered a path e_i from below use'$_i$(x), since this would imply that it moved above use'$_i$(x), violating dependency #2, or again dependency #4.
- Def(x) can't be *outside* the code modified by the trace scheduling transformations, and have use'$_i$(x) be reached by it as a result of compaction, since on any equivalent path all operations in the original path are preserved during compaction, and dependencies between operations are also preserved. Thus if use'$_i$(x) is reached by def(x) on some e_i in P', which is an equivalent path of some p in P, Then def(x) must also have reached use(x) in p (or else e_i does not satisfy the requirements of correctness, contradicting the results of the previous chapter.)

Thus our supposition is false, and no such def(x) can exist, and therefore for all def(x)$\in U_{i=1,n}DEFS'_i(x)$, def(x) is also in DEFS(x), thus $U_{i=1,n}DEFS'_i(x) \subseteq DEFS(x)$.

(b) $DEFS(x)' \subseteq U_{i=1,n}DEFS'_i(x)$

Suppose the above weren't true. then, there must exist a $def(x) \in DEFS(x)$ which is not in $\cup_{i=1,n} DEFS'_i(x)$. Then:

- If def(x) is in the modified code, then there must have existed at least one trace-path through the original trace in which def(x) reached use(x). If no such path exists in the new program graph P', this would violate the (partial) correctness of trace scheduling.
- If def(x) is outside the modified code, either some "new" definition would have had to move between def(x) and (all) $use'_i(x)$, in P', blocking def(x) from reaching an use of x, or use(x) must itself not appear on any equivalent paths in P', when it did appear on at least one trace-path in P (or else def(x) would not be part of DEFS(x)). Both of these situations are impossible, without violating trace scheduling correctness (all operations in a trace-path are preserved by equivalence paths, new operations preserve dependencies and no operation can move into a path if it writes a variable which is live on that path. Thus our supposition is false, and no such def(x) can exist, and therefore for all $def(x) \in DEFS(x)$, def(x) is also in $\cup_{i=1,n} DEFS'_i(x)$, thus $DEFS(x) \subseteq \cup_{i=1,n} DEFS'_i(x)$.

Thus both (a) and (b) above must hold, or else contradictions occur. Therefore $DEFS(x) \equiv \cup_{i=1,n} DEFS'_i(x)$, implying that the correctness of static reach analysis is preserved, despite the trace scheduling transformations. □

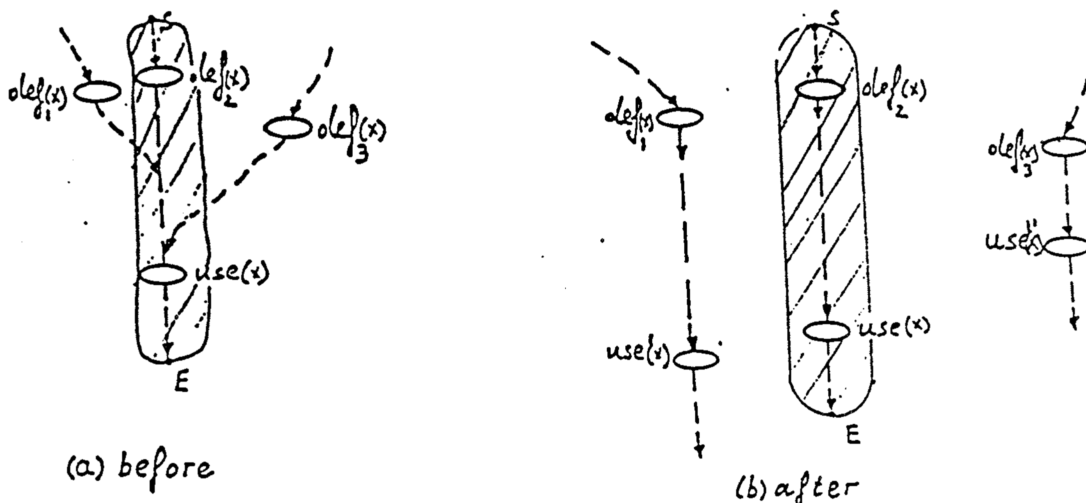


Figure 5-2: Reaching definitions are conservative
(a) Before Compaction
(b) After Compaction

Note that the fact that static reach analysis is correct does not necessarily mean that it is also very accurate. In fact it turns out that in certain cases it may be a little over-conservative, as the example of figure 5-2 shows. Two trace-paths which shared some operations before compaction may well be totally separated as a result of the application of trace scheduling. Because of this, a use of some variable x, which was originally shared by the two paths will be separated (copied) on each new one. This may mean that fewer definitions of x reach any of the uses after compaction than before (even though the union of the reaching definitions will be the same). In turn, this could allow for finer shades of discrimination in the disambiguation process. For example, if x is an index in an indirect reference, it may be possible to decide that no conflict between the operation using x and some other operation can occur in the compacted program. This can be so even if this decision was impossible in the original program (due to the sharing of operations between paths). Since this effect did not seem to be noticeable in our experimentation with the BULLDOG compiler to date, and since the correctness of the disambiguation can be maintained even with static reach analysis, we haven't pursued this matter any further. However, as will become clear in the process of discussing live-dead analysis, *local* updates between trace compactions would be sufficient to ensure the elimination of this conservative effect of static reach analysis, while still preserving correctness.

5.2 Live-Dead Analysis for Trace Scheduling

Unlike reach analysis, live-dead analysis is used by the disambiguator (and the trace scheduler through it) to obtain information about variables being live **at a certain point**. As we said above, two "corresponding" points in the uncompacted and compacted programs will not necessarily be "equivalent" in terms of the flow information being identical.

A case in point is that of split points for the purposes of conditional jumps blocking code movements in accordance with dependency #4. That completely static live-dead

information cannot be always used correctly, because live-dead information may change during compaction of a trace, can be seen from the example in figures 5-3 and 5-4. As a result of compaction, operation (*) has been copied as split compensation, and has lead to the change of the live-dead information at the split point of the conditional jump (see figure 5-4).

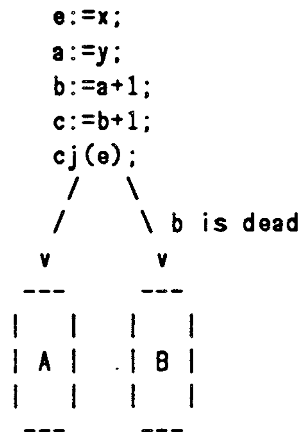


Figure 5-3: Live-dead information before compaction

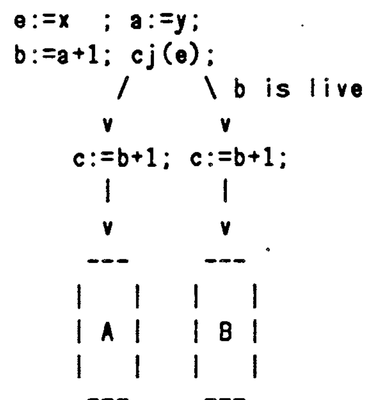


Figure 5-4: Live-dead information after compaction

The reason the live-dead information changes at these points is that operations which were above the conditional jump are, as a result of the conditional jump's move, injected into the split by the split compensation mechanism. Note that the movement of operations from below the conditional jump can't influence the live dead information on the off-trace branch, and neither can the rejoin compensation since all the operations which get moved into the rejoining branch as rejoin compensation, or are below the rejoin as a result of

compaction, used to be on that path (and in the same order) in any case, or else there would be a violation of the requirements for trace scheduling correctness.

It would appear that the changes introduced by the compaction make the static live-dead analysis worthless for our purposes, or that at the very least we will need extended updating for each split (i.e. full propagation of the changes, throughout the program, until the def/use sets settle down again). Despite this we were able to demonstrate that static live-dead information can be directly used during the trace compaction itself. Before we can do this however, we will have to do some preliminary work.

Lemma 7:[26] Live-Dead information at the Entrance/Exit points before and after the compaction of a trace (i.e. in P and P') is identical[27].

Proof: The program-graph is only changed by the compaction of the trace and thus the only modifications appear in between entrance and exit points. All operations on any trace-path are preserved and since dependencies are also preserved on any trace-path, the relative order of defs/uses of the same variables on any given trace-path is also preserved. Thus the live-dead information is unchanged at entrance/exit points (and thus in the rest of the outside program graph as well).

Suppose this were not the case, then on some trace-path in P' we must have either missing operations with respect to P or extra operations, or some def/use pair must have reversed its order (if none of this happens, then live-dead information in P' *is* unchanged at the entry/exit points, with respect to that in P). But each of these conditions contradict the requirements for trace scheduling correctness, so they cannot be satisfied. □

---

[26]Essentially the same argument made in this proof would apply equally well to reach analysis. Therefore a local periodic update of reach information, similar to the one suggested for live-dead in theorem 12 would be adequate for our needs. Since we have already seen that even static reach information is correct for disambiguation, updating reach analysis only after each trace compaction is clearly not a concern.

[27]Note that this lemma does not imply that the live-dead information at the rejoin/split points doesn't change (in fact it does change, as we will see presently). What it does imply is that updated live-dead information can be obtained by local updates after each trace compaction, without any loss of accuracy with respect to a fully dynamic analysis.

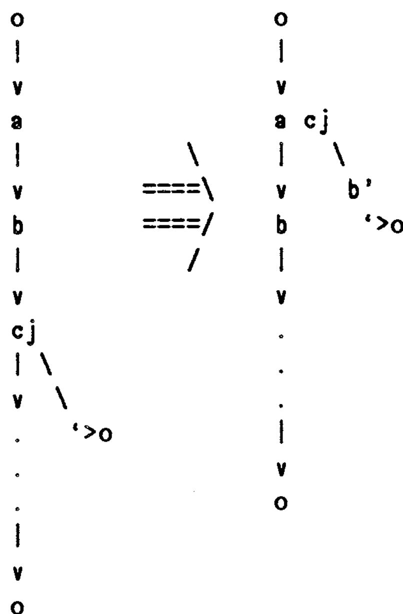


Figure 5-5: Live-dead Information During Compaction
(a) Before Compaction
(b) After Compaction:
The effect of b' on live-dead information
off-trace is nullified by b on trace.

Theorem 11: No dynamic updating of live-dead information is needed **during** the compaction of a given trace.

Proof: By lemma 7, live-dead information may not change at the entrance and exit points in P' with respect to those in P. In particular, the information at a split-point S_i is the same before and after the compaction of the trace. Thus any change in live-dead information at the actual split, must be due to split compensation.

However, operations in split compensation that may change live-dead information appear below the conditional jump causing the split in the compacted trace body, and thus block any operation which would potentially violate dependency #4 (as a result of not having dynamically updated live-dead). Thus such operations are not even allowed to reach the conditional jump (see figure 5-5). Note that this refers to both operations which would increment the live-dead set at the split, as well as to operations which would decrement it (i.e. both uses and definitions). □

While the above theorem enables us not to worry about continuously updating live-dead

information during the compaction of each trace, and thus avoids the need for truly dynamic updating, we have seen that live-dead information can and in fact does change during each trace compaction. We will now present a simple example where such changes could in fact lead to incorrect trace compactions.

Trace Scheduling only uses live-dead information for the purposes of resolving dependency #4 during trace compaction. The only concern is then that live-dead information may change during TS at the split points, and thus incorrect code motions may result.

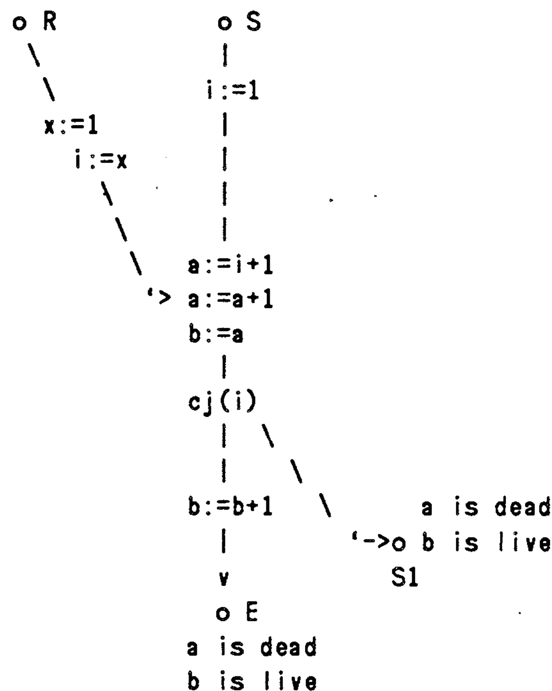


Figure 5-6: Incorrect Trace Compaction as a Result of Erroneous Live-dead Information)
(a) Before compaction of trace S->E.

For example in figure 5-6, the original trace being compacted is S->E. After the compaction, we obtain the graph in figure 5-7. If in this new graph we choose to compact the trace-path R->(*)->S1, the static (original) live-dead information (on the branches of the conditional jump) will not prevent operation (*) from moving across the conditional jump, which is incorrect; that is, a violation of the requirements for trace scheduling correctness will result: the statement "a=a+1" will appear twice on one of the equivalent

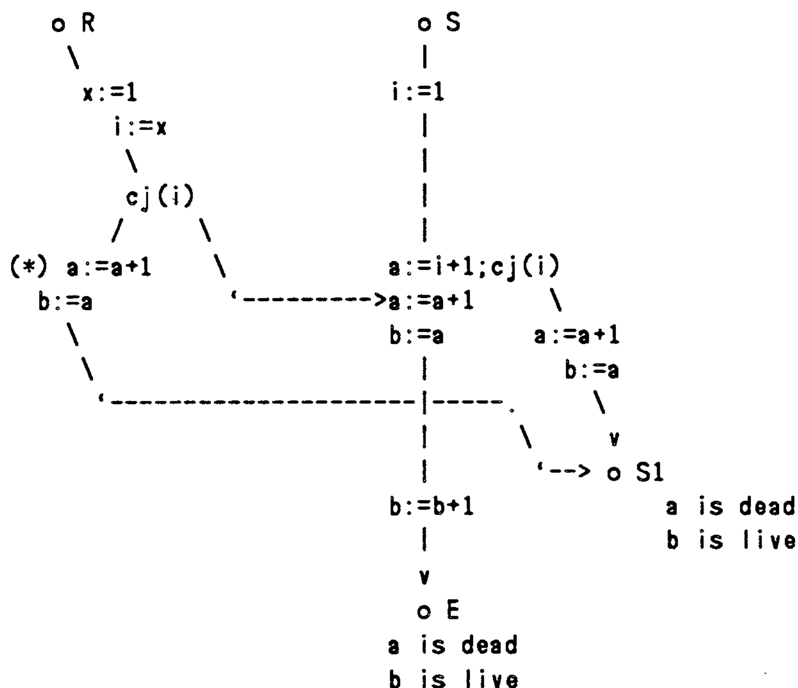


Figure 5-7: Incorrect Trace Compaction as a Result of Erroneous Live-dead Information)
(b) Before compaction of trace R->(*)->S1

paths of R->S1, while the original trace-path R->S1 contains it only once. However, we will show that the static information coupled with incremental *local* updating is both correct and non-conservative for the purposes of trace scheduling. The next theorem will show this.

Theorem 12: Incremental updating of live-dead information is both correct and accurate.

Proof: By lemma 7 it follows that the obvious one-pass local updating - propagating from the exit points of the trace where the live-dead information is unchanged (and therefore known) upwards to the entry points where live-dead information is again unchanged (and therefore no need to continue throughout the program graph) - is correct, if indeed the new information which we propagate to the entry points is identical to the original one there. This later fact is guaranteed by the correctness of trace scheduling, which ensures that no operations have disappeared from between an entry point and an exit point, that dependencies have been preserved and that any new

operations don't violate dependencies, thus preserving def/use chains.

Furthermore this is as accurate as fully dynamic analysis since by lemma 7, after each trace live-dead information can be fully updated by this local mechanism (since the changes are only local) and by the argument in theorem 11 in between updates (i.e. during trace compaction) the difference between such local updates and fully dynamic analysis will not affect the results of the compaction. □

5.3 Live-Dead Analysis for Finer Granularity of Array References

This section conceptually belongs with the improvements to the disambiguator more than with the flow-analysis correctness, since it is less a matter of proof than of simply enhancing the efficiency of the disambiguator. However the ideas presented in this section appeared (chronologically) only after the other flow analysis issues were settled and well understood. Before the formalization of trace scheduling was completed and the ensuing analysis well understood, the whole concept of live-dead analysis for trace scheduling in general was too fuzzy in our minds to attempt any refinement for better handling of arrays. Thus we believe that this section really has its place in this chapter.

5.3.1 The Problem

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. For example, in figure 5-8, 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 alive 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[28].

The simplest approach to improving the accuracy of array live-dead analysis is to propagate *each element of an array as a separate variable,* and use the disambiguator in deciding what gets defined or used each time an operation involving array references is encountered. This in a sense would be the most effective way of improving accuracy, since it would provide the finest level of granularity we could possibly achieve. Unfortunately both the space and time requirements of this scheme would make any realistic implementation ridiculously expensive.

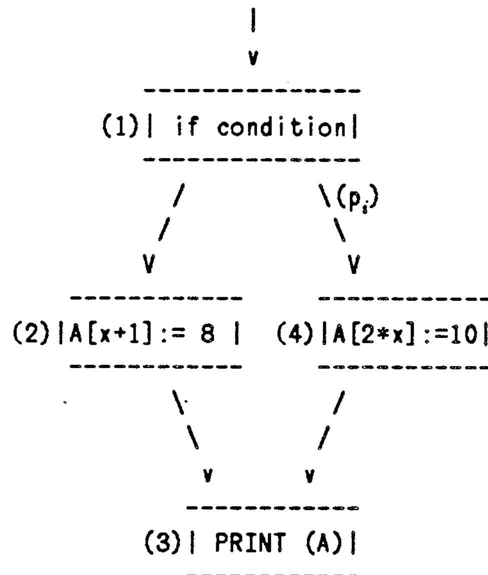
A slightly more complex but more efficient approach is to propagate *the indices together with the array names, in effect considering each reference as a separate variable,* but that't not enough; neither is just a range of values, since they may not always be able to express accurately what is live at a given point: for example in figure 5-8 saying that a[2*x] is not live at p_1 is irrelevant if $x+1 \neq 2*X$. In that case, statement (2) may or may not be able to move above statement (1) (during trace compaction), depending on what else is live at p_1[29]. To ensure correctness, the conservative way out must be taken when in doubt.

[28]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.

[29]If A were not live at the end of the program segment in our example, statement (2) *may* be allowed to move above statement (1), regardless of whether $x+1 \neq 2*x$ or not; if *A* is live, then it may be illegal to move statement (2), unless $x+1=2*x$.

5.3.2 Our Solution

What we really need is a live-dead analyzer with a fully integrated disambiguator, and a finer discrimination than just subranges. Thus for example, consider again the case in figure 5-8.



Can statement (2) in the trace 1,2,3 move above statement 1?

Figure 5-8: The Live-dead problem for Arrays in Trace Scheduling

To be able to answer the question in figure 5-8 with a yes, we would need a lot more than a range. Since statement 3 wants to "USE" the whole array A, this is what is to be propagated backwards (live-dead being a backward flow problem). Statement 4 should then "DEF" only those elements of A which are of the form 2*x (and if some range on the values of x can be determined, this could be refined further say to "even between 2 and 10 and between 26 and 36, not including 30". Thus the refined live information pertinent to array A at the top of 4 should be "all elements of A which are not '"even between 2 and 10 and between 26 and 36, not including 30"'.

Thus 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:

- 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.

While the above may not be simple to implement, it may well be worth doing. It may even be useful for refining the reach analysis, particularly if range information is available.

To improve efficiency, live-dead analysis could be done in two phases, one for scalars, (which could still use bit vectors) and a more complex but hopefully shorter one just for vector references using the method described above.

5.3.3 Implementation

The approach described above has never been fully implemented. A simpler version of it has been experimentally implemented. In it the actual definitions of array references are used in a dead propagation. This still allows bit-vectors to be used, since each reference may be considered as a separate variable. Still, this approach suffers from the same problems mentioned above. It has to err on the conservative side to achieve correctness, it has to assume that operations cannot move above a conditional statement whenever the reference in the operations are not totally equal to the ones in the dead set on the off-trace branch. While this is not very different from the live propagation scheme, it works better in practice since full arrays are often used on output (e.g. printed), which tends to inhibit the earlier schemes, rendering them more restrictive that this one. Furthermore, as a result of compensation, copies of operations will often appear on both branches of a conditional statement. This will also improve the effectiveness of our scheme with respect to the conventional live information.

To increase accuracy beyond this point, some real symbolic manipulation of the kind presented above seems unavoidable, but more practical experimentation is needed to decide whether the potential gains are worth the extra costs in terms of time and complexity.

Chapter 6

RUNTIME DISAMBIGUATION

6.1 The Idea of Runtime Disambiguation

Runtime disambiguation (RTD) treats memory anti-aliasing of references which cannot be effectively disambiguated at compile time (e.g. an array index as a function of some input variable). Using it, part of the disambiguation mechanism is integrated into the parallel code produced by the compiler.

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

Figure 6-1: Original code before RTD code insertion

Given two references on a given trace which cannot be disambiguated at compile time a potential conflict between them (i.e. $i=j$), (figure 6-1) has to be conservatively assumed to ensure correct handling using the traditional disambiguation approach. Runtime disambiguation, on the other hand will transform the code segment to that in figure 6-2.

The .9 probability estimate gives the trace (a,l1,b) priority in the compaction process. The assertion in statement l1 will be placed in a small data-base for the disambiguator and will supercede any information the disambiguator may have obtained about the references

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

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

Figure 6-2: Code after RTD code insertion

for that path. This information will be used as a compiler directive, allowing a,b to be scheduled independently. Of course, the off-trace branch must take care *at runtime* of the case in which $i=j$. Depending only on the computation of i and j, the *if* statement might be placed early in the schedule (quite possibly for "free" if resources are available). So if the conflict between the references is either nonexistent (but the disambiguator cannot establish that by itself at compile time) or occurs rarely (e.g. i,j are used to traverse A in opposite directions), the potential speedup resulting from the use of RTD can be significant.

6.2 Practical Concerns

While the above describes the essential idea of runtime memory anti-aliasing, the actual implementation requires several considerations which are discussed bellow.

6.2.1 Renaming

While the inserted code may move during compaction, initially it must appear immediately before (b) to insure that i and j have both been computed. The use of the *single assignment principle* by which variables are only assigned to once, will greatly alleviate the concern of the initial place where the RTD code can safely be inserted. Otherwise, cases may arise where no legal place can be found for the insertion of RTD code, due to the reuse of index names. Thus for example no place can be found in the code fragment of figure 6-3, where RTD code could be inserted, since at no point in that program are both indexes available for comparison. To avoid such situations renaming is critical.

An additional reason for renaming to be used is that it may significantly increase the

```
                i:= read();
(a)             A[i]
                i:=read();


                j:=read();
(b)         l2: A[j]
```

Figure 6-3: Impossible RTD code insertion

potential parallelism found using runtime disambiguation. While renaming is often used in compilers in an attempt to eliminate spurious dependencies and thus increase the potential for parallelism exploitation, its use is particularly important in the context of RTD, since without it the RTD code may not be compactible and thus may actually slow down the computation due to the extra overhead introduced.

6.2.2 Correctness Considerations

Assertions are used as compiler directives, not as real operations. These assertions force different types of compaction on each branch of the *if* operation. These assertions will alter dependencies based on assumptions that can only be checked at run time. Therefore, we must ensure that the (compile time) compaction does not produce code that cannot recover if the assumptions prove to be incorrect. Unrecoverable code may be created if both (a) and (b) are scheduled before the *if* operation that checks the assertion because A[i=j] may have already been assigned a wrong value or used in the wrong way on the assumption that $i \neq j$.

There are several cases in which this situation can occur. In figure 6-4 we can see several in which no recovery is possible. It may appear that the situation is complicated further by the interaction of multiple conditional jumps as part of separate RTD segments. However, the normal trace scheduling compensations will suffice to handle such situations, in the absence of the problems mentioned previously. This is due to the fact that the only way RTD tests could interact is by somehow avoiding (e.g. skipping over) some other RTD test, and reaching in this way some potentially illegal (since based on an unverified assumption) compacted code. However the correctness of trace scheduling precludes this from

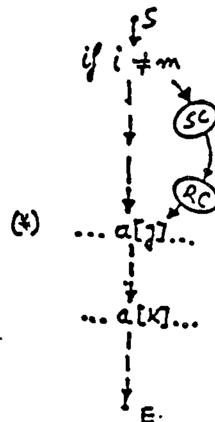
```
   read(x,i);
   a[i]:=x;
   a[2]:=2;
   print(a);

     ||inserting RTD
     ||
     \/              incorrect
   read(x,i);        compaction      read(x,i); a[2]:=2;
   a[i]:=x;            ---\          a[i]:=x; if i≠2 l1,l2;
   if i≠2 l1,l2;       ---/    l1,l2:print(a);
l1:assert i≠2;
l2:a[2]:=2;
   print(a);
```

When i==2 we get: a[2]==a[i]==2 in the original code,

and a[2]==a[i]==x in the compacted code.

Figure 6-4: Sample problematic compactions using RTD



Where could "if j≠k" occur?

- If bellow (*), a[j] is out of it's scope (reducing to case above).
- If above (*), the *if* will be copied as RC compensation and get correct results.

Figure 6-5: Sample compactions involving multiple *if*s created by RTD

happening, since it ensures that all operations are preserved on all equivalent trace-paths during compaction. In particular, any *if*s which are skipped over during compaction will be part of compensation (see figure 6-5). Therefore we only have to worry about individual

RTD segments. Two solutions can ensure that correctness is preserved in the face of the above problems.

The first solution is to never allow operation (b) to move at or ahead of the test; conservatively adjust the new rejoin point of the test to be after both (a) and (b) in the trace, thus forcing both to be copied (in original order) as rejoin compensation. In figure 6-4 this would mean that operation (b) could not move above the *if*. Although this solution is very simple to follow and implement, it may limit the code movement and thus the benefits of RTD. While this is not desirable, it is not necessarily a severe limitation for RTD, since the *if* itself is very likely to move upwards in the schedule during compaction (being highly compactible) and thus in general won't hinder the movement of the operations it governs.

The second more general solution introduces compensation to restore the correctness of the compacted code. Such a solution has the dual appeal of not restricting code motion unnecessarily (thus allowing finer granularity) and resulting in a natural extension of trace scheduling which allows the correct handling of RTD. In this scheme practically unrestricted code movements are allowed. The trace scheduling compensations for RTD tests are modified so as to provide adequate compensation, regardless of whether the test was reached before or after the operations governed by it were executed. To do this the transformations must be prepared to provide *undo* type compensations, as that in the example of figure 6-6.

In this example, the false branch must not only undo the execution of statements (a) and (b) which will have been executed in the wrong order, (when $i=j$) but also any operations in code block #1 which may have been affected by the execution of (a) and (b) in the wrong order. This can be quite hard, and may not even be realistically feasible, as in the case of statements involving the input of information (e.g. from a stream) which cannot be reversed in general. An additional problem is that conditional jumps may be affected by the results of statements (a) and (b) complicating the state restoration. Furthermore the overhead incurred in terms of new code generated as part of this new type of compensation

may offset any gains achieved by RTD. A more careful analysis of the tradeoffs involved and more experimentation than what we have done to date are needed to determine the effectiveness of this approach.

```
(b)      A[j] := x;
(a)      y := A[i];

         [Code block #1]

         if-ine i j  l1 l3 .9
    l1: assert i≠j

    l2:

         ..........
         Stop
    l3: Restore{A[j], y}
         Goto l2
```

Figure 6-6: Requirements for Undoing Mechanism

6.2.3 Space Considerations

In the process of inserting RTD code, we will increase the number of potential traces in the code because of the extra conditionals introduced. This may result in a code explosion in the compacted program. This is due to the high mobility (data independence) of the introduced conditionals, which will result in having them be propagated as part of the various compensations during trace scheduling. This will creating even more new paths, which in turn have There are to be compacted. Since RTD is just another optimization, we have to effectively limit this code explosion for it to be practically usable without unduly slowing down the compilation process and incurring unacceptable space costs. There are a number of ways in which we can satisfactorily restrain the space explosion without drastically reducing the potential benefits of Runtime Disambiguation.

6.2.3.1 RTD as Dynamic Assertions

To start with, even if we choose not to use RTD in general, the technique can still be applied as an enhancement of the assertion mechanism (a kind of "when $i \neq j$ assert..."), even in cases where straightforward assertions are not applicable, as in the example of figure 6-7.

```
    for i:= m  to n  by l
        j:= m1 to n1 by l1  do
        {unroll x}
    (*) A[i]:=  A[j]
        end;
```

Figure 6-7: Use of RTD as a dynamic assertion

We cannot use simple assertions to allow versions of statement (*) to be executed in parallel, since there may sometimes be a conflict between the different versions of (*) from the different loop iterations. A significant speedup could be achieved, however, by directing the compiler to insert RTD code if the conflicts are relatively infrequent. By letting the user have control over when to insert such *dynamic assertions*, the space increase will be minimal, and may result in a significant speedup. This may also be viewed as an improvement over simple assertions from the user's view point, since its use does not require a fundamental understanding of the algorithm being compacted. Thus a naive user would not have to precisely know whether two ambiguous references may never conflict; he may simply think that it would be potentially beneficiary (in terms of the speedups achieved) if they actually didn't.

6.2.3.2 Smart Compiler Generated RTD

A more sophisticated approach to the space problem is to go one step further and eliminate the user's input. This can be done by letting the compiler insert RTD code only when it is certain this will help reduce the schedule length (i.e. time). This might require two-pass trace scheduling in which the results of the first pass direct the insertion of RTD code in the second. That is, only in cases where the conflict between two ambiguous references will have inhibited compaction during the first pass will RTD be inserted.

6.2.3.3 Standard Optimizations

Standard techniques such as common subexpression elimination can also be used to reduce duplication of RTD code. These techniques will be particularly effective with unrolled loops, the type of code that interests us most. Parallelism need not be adversely affected by these techniques because we will only eliminate identical index calculations which need not be repeated.

Peephole optimizations can be used in two ways to alleviate the space problem. First, when multiple vector operations with potential dependencies exist, the situation shown in figure 6-8 will tend to occur as a result of RTD code insertion.

```
(a0)   A[i0]
           ........
(a1)   A[i1]
           ........
           ........
(a2)   A[in]
           ........
           ........
           if-ine i0 j .9  l1  l2
       l1: assert i0≠j
       l2: if-ine i1 j .9  l3  l4
       l3: assert i1≠j
       l4: ........................

       lk: if-ine in j .9   ll   lm
       ll: assert in≠j
(b)    lm: A[j]
```

Figure 6-8: Clustering of multiple RTD

This can be converted into a semantically equivalent segment containing only one conditional jump, as shown in figure 6-9.

This approach inhibits parallelism by creating dependency chains in the calculation of t_n, the aggregate test variable. Using a binary tree approach to calculate t_n will alleviate this problem. The significance of this disadvantage is reduced for very long traces in which the dependency chain for t_n may be much shorter than the overall schedule, and thus may not be a determinant factor in the actual length of the schedule for the trace.

The second peephole optimization which can be applied to reduce code proliferation is

```
(a0)   A[i0]
              ........
(a1)   A[i1]
              ........
              ........
(a2)   A[in]
              ........
              ........
              ine i0 j
              ine i1 j
              iand t1 i0 i1 j
              ..................

       lk: if-ine tn T  .9  ll  lm
           assert i0≠j
           assert i1≠j
           ..............
       ll: assert in≠j
(b)    lm: A[j]
```

Figure 6-9: Assertion tests unification

possible because a substantial part of the code generated as compensation for RTD *if* tests will be identical and can be shared.

6.2.3.4 RTD Code Elimination

Perhaps most important, we can reduce the code explosion dramatically by realizing that the extra code introduced for RTD has no significance (other than modifying compaction on some traces. Thus at any point during compaction we may simply choose to eliminate the copying of RTD code in rejoin/split compensation and the resulting program will be unaffected. This choice will significantly reduce the size of the code produced since no code explosion (due to RTD) will result if there are no copied (RTD) conditionals. In addition, the elimination of the extra code by itself will contribute to reducing the code generated. Furthermore, if the trace-picker does a good job in selecting traces in order of their likelihood of being executed, removing the RTD code from all but the first few traces should not significantly affect the speedup resulting from RTD.

Notice that this will require a slightly modified rejoin compensation to preserve correctness. This is a result of preventing RTD code from being copied as part of rejoin compensation, which may lead to problems, as illustrated in figure 6-10.

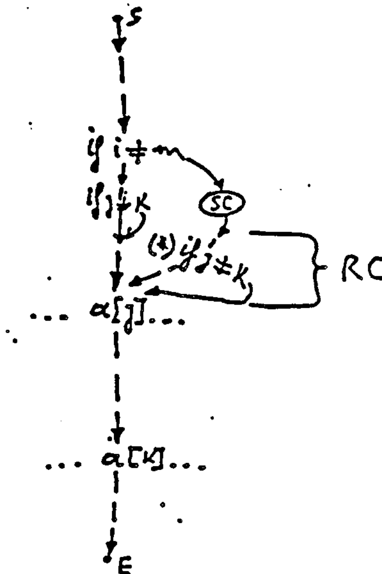


Figure 6-10: Eliminating copied RTD code from compensations

Eliminating the copied test (*) from the rejoin compensation for the upper *if* creates a path on which (a) and (b) are outside the scope of their governing test. The compensation which would remedy this situation, in the absence of the *undo*-ing mechanism, requires that the new rejoin point r_p for a RTD test in the trace body be at a MI_k such that:

- Only operations originally below the rejoin are below r_p in the compacted code, and
- r_p can be above vector operations which can't be disambiguated at compile time only if these vector operations are in dependence order (i.e. "a precedes b" or "a not lower than b").

This in effect forces the rejoin compensation to contain the indirect references whenever they may escape the scope of a governing test (which wasn't copied as compensation being only RTD code). Since copied operations will be in original order, this compensation will ensure correctness, by providing an effective escape path.

6.3 Implementation

6.3.1 Modifying BULLDOG

To implement the runtime disambiguation system we had to modify the BULLDOG compiler in several ways[30].

First and most important, we had to decide *where* the runtime disambiguation code had to be inserted. Inserting such code in the absence of any other information wouldn't do, since it would mean that code would be inserted for each pair of indirect references in our program. Obviously this would be too frivolous in terms of both space explosion and processing time. To avoid this, we modified the compiler to do two passes. During the first pass, the input program is compiled into sequential NADDR and reaching and other analysis used in the process of disambiguation is computed. Then, each *potential* trace in the program is examined, and along each one indirect references are compared. Those which may be ambiguous are then selected for runtime disambiguation, an appropriate place where the RTD code may be inserted is located for each such pair, and the code is actually "injected" into the original sequential NADDR code.

In the second pass, the initial analysis is redone, to enable the analysis to take into account the added information[31]. Then the process of trace scheduling proceeds as usual, traces being picked and compacted. The conditional jumps (*if's*) inserted as part of RTD code are treated as any other normal operation. The actual assertions require further modification of the compiler for runtime disambiguation, despite the existence of an assertion system implemented by J. Ellis for the BULLDOG compiler. The original system simply stores assertions in a data-base available to the disambiguation module and which is checked whenever two indirect memory references are compared. The information in this

---

[30]Since this is a very experimental module, which will require more investigation, it is not an "official" part of the BULLDOG compiler, and was only added at this stage on a separate version, for the purposes of our experimentation.

[31]This is required mainly as a result of the use of unique pointers to represent each operation, which is in turn used for efficiency reasons. In a production implementation both passes done here could be combined into one. In fact, this would be also desirable in order to achieve greater accuracy in the picking of potential traces.

data-base overrides any decision that the disambiguator may make, and is ***global***, over the whole program. This last feature has a severe drawback for our purposes: it may allow precisely the unrestricted type of code motions discussed in the previous section which may lead to incorrect compaction if used as part of RTD[32].

The simplest way to circumvent this, is to disable the normal updating of the assertion data-base, and to add a mechanism which will, at the beginning of the compaction of a trace, insert RTD assertions for that trace dynamically. An other mechanism will then remove the assertions from the data-base after the compaction of the trace is completed. In this way the RTD assertions will apply only on the traces where they were introduced. Finally we need to disallow code motions beyond the scope of the tests governing the assertions within the trace. Alternatively, adequate compensation, providing an escape route at runtime which will restore correctness is required. These two methods of ensuring correctness have been discussed in the previous section. Of these only the first was implemented.

6.3.2 Correctness and Space Considerations

As we mentioned in the previous section, we ensure correctness by never allowing an operation to move at or ahead of the test governing an assertion which affects it. In addition we have provided the modified rejoin compensation which enables us to implement the elimination of RTD from compensation code and unimportant traces, while maintaining correctness (see section 6.2.3.4).

The space saving methods we have implemented include the peephole optimizations discussed in the previous section and RTD code elimination, with its associated modified rejoin compensation. We can also take advantage of the existing Common Subexpression Elimination system in the BULLDOG compiler. Indeed, programs which could not be run through our modified compiler without CSE due to the code proliferation, were quite

---

[32]That is, it will allow code governed by the assertion unrestricted movement outside the scope of the test checking the correctness of the assertion.

manageable when CSE was turned on. Furthermore in our (limited) experience, CSE didn't seem to negatively affect parallelism. All the space saving methods are available as options to the user, with the default being total RTD. The peephole optimizations can be activated by setting a flag. The number of traces after which RTD generation is to be suppressed (for RTD code elimination) is indicated by the value of another global flag. The details of the implementation, are given in appendix III. This appendix contains both the original BULLDOG routines modified for RTD, and the additional routines implementing the runtime disambiguation proper.

6.3.3 Experience with RTD

The version of the BULLDOG compiler which we used for experimentation with RTD was written in MACLISP, which does not provide virtual memory management. Given the size of the compiler itself, the available working memory was less than 150k words. This has heavily restricted our ability to experiment with realistic programs in realistic situations. Indeed experimentation with large programs in general (not only for RTD) was severely hindered by this problem[33].

Program	Total RTD (Speedup)	Space RTD (Speedup)	1 Trace RTD (Speedup)
FFT	16.5%	NEG.	16.5%
INVERT	77%	0	77%
UNION	50%	NEG.	50%

Figure 6-11: Preliminary results - RTD

[33]Ultimately the compiler was ported to ELISP, a variation of Rutgers Lisp which provides extended addressing.

Because of these problems, the results which we obtained are not as good as they may have been, or as numerous. While still encouraging, we could do much better if more unwinding were allowed. Indeed, a cursory look at the compacted programs indicates that the reason why more dramatic speedups were not achieved is that at the level of unwinding we were able to do (typically 2) the length of the schedule of the average trace is still dominated by the computation of indexes and other arithmetic expressions. Under such conditions disambiguation (runtime or not) will always have less impact. This is particularly acute in the case of assertion tests unification, in which the length of the computation of the test expressions may often (in small traces) be the critical factor in the length of the compacted schedule. Thus we expect that much better results could be obtained by RTD with the new BULLDOG compiler. The very small sample given in figure 6-11 which was obtained with the original compiler, is therefore just meant to give a preliminary flavor, and to show feasibility rather than an overwhelming empirical proof of success. As such we will not attempt to draw any conclusion, before we investigate the issues of runtime disambiguation more fully.

Chapter 7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

The experience which was gathered in the development of the experimental BULLDOG compiler and the experimental results obtained constitute strong evidence that Trace Scheduling in conjunction with memory anti-aliasing tools is effective in the exploitation of parallelism for VLIW architectures[34].

As a byproduct of our efforts to design efficient flow analysis methods appropriate for disambiguation and trace scheduling we were able to show that Trace scheduling is correct and terminates. Its effectiveness could possibly be improved at the expense of more complexity and space/time tradeoffs. We have identified several alternative transformations which could provide improved performance.

Efficient flow-analysis methods can be used for disambiguation and trace scheduling. We have described these methods and showed their correctness in the context of a trace scheduling compiler.

Finally, the preliminary evidence which we have gathered to date seems to indicate that

---

[34]While the disambiguation and flow analysis mechanisms described in this thesis are critical for the effectiveness of our methods, several other issues are equally important. These have been treated in detail in [9], [33].

runtime disambiguation can improve the compactor's ability to deal with code which has unpredictable memory references. This would enhance the ability of trace scheduling compilers to perform well on a larger variety of code than currently possible.

7.2 Directions for Further Research

As a result of the work for this thesis several directions for continued research have become apparent. Some of these are just natural extensions of this work, while others are significant departures, based on the experience we have gained in the process:

- Applications of Trace Scheduling/Disambiguation techniques to other languages or domains.
- Applications of Trace Scheduling/disambiguation techniques to other architectures (e.g. multiprocessors, lookahead processors).
- Runtime Disambiguation refinements and variations.
- Trace Scheduling transformations refinements and variations.
- Identifying a small set of primitive transformations subsumming TS, and possibly other optimizing transformations.

7.2.1 Beyond the ELI

Large amounts of parallelism are potentially available in ordinary programs [29]. The Bulldog compiler attempts to use trace scheduling to exploit this parallelism on VLIW (e.g. ELI) machines. We have demonstrated that this approach can be successful. However based on the insight we have gained in the process we believe that we can do better yet. To understand this, let us reiterate the main properties of the approach:

Advantages of a Trace Scheduling Compiler:

- Global compaction (i.e. beyond basic blocks).
- Uniformity (i.e. not case-by-case handling of different types of code).
- Graceful degradation (adaptability to variations in number of processing units)[35].
- Fine level of granularity of parallelism.

While all these qualities are extremely desirable, there are several drawbacks to the process of achieving them, which limit the practicality of the pure BULLDOG approach to (a very important subgroup) of applications: scientific programs. These disadvantages are:

1. Expensive (time, space) compilation, due to the level of granularity at which the compiler operates.

[35]Albeit at the expense of recompilation.

2. Performance is critically dependent on predictability of conditional jumps, due to the necessity to pick traces.
3. Very sensitive to the ability to do successful disambiguation (both memory and bank), based on the empirical fact that often large amounts of parallelism are found in code involving indirect references.
4. Relatively localized detection and exploitation of parallelism, resulting from the inability to pick traces across loops (unless they are fully unwound).

1,2,3 above reduce applicability of trace scheduling compilers (e.g. to scientific code). 2,3,4 reduce the potential for full parallelism exploitation.

As for the ELI itself:

Advantages:

- Units can perform different operations (more versatile than array processor).
- Very limited runtime synchronization costs.

Disadvantages:

1. Absolute reliance on compile time scheduling may be too conservative.
2. Not significantly scalable.

While reliance on compile time scheduling eliminates synchronization problems that have traditionally plagued multi-processors, some flexibility in this area may prove profitable (e.g. for a parallel sorting program, for example).

Scalability while unimportant at the ELI level becomes critical if yet larger amounts of parallelism are to be exploited (e.g. if we could overcome problem 4 above). The problem is even more acute when multi-processing is considered. Inability to scale up significantly is a result of 1-4 above.

Note that this is not a critique of the ELI. The goals which it has set for itself are realistic and totally achievable. In that context none of issues which are mentioned here as drawbacks are critical. It is only when trying to go beyond these goals that these issues become important. As such, this should be viewed as an attempt to extend the results obtained by the ELI project.

We would like to exploit as much parallelism as possible, over and beyond a single group of nested loops, or even a single program. We would like to maintain the advantages of the

ELI/BULLDOG approach while overcoming some of the drawbacks. We believe this can be achieved by adopting a hierarchical approach both at the compiler and architectural level, to differentiate between fine grained versus global parallelism[36].

[36]The notion of global parallelism here is similar to that of Padua [30], and the Cedar project in general. The approach, however will be quite different.

Appendix I. The Parafrase System

I.1 Goals

While the different modules that form Parafrase can be used for several purposes (even to increase locality of reference in virtual memory systems), their main goal is to generate code for fast "highly parallel" machines (e.g. vector, pipeline and multiprocessors). This is done by a series of **source level** transformations of a general nature, as well as more hardware oriented transformations. The main argument for source to source optimization is portability - lets the compiler supplied with the machine do the actual vectorizing.

In general the system assumes the following type of hardware:

- Vector registers
- Sophisticated indexing hardware
- Chaining (overlapping) functional units
- Mode/mask vectors instead of conditionals
- Vector instructions

I.2 Outline of the System

The modules of Parafrase are hooked together (sometimes the order in which this is done has to be determined "empirically") to perform a series of transformations, which will result in a semantically equivalent program with more explicit parallelism. Some of the modules can be used for various other optimizations, such as memory management (caching, paging and improving locality of reference). Since this may be contradictory to parallelism exploitation, there is clearly a trade-off involved.

The sequence of optimizations done by the system is:

I.3 Frontend

(Used For General Parallel Architectures):

- **Induction variable substitution** (remove superfluous index variables - which are linear functions of the loop indices.
- **Renaming (for scalars).**
- **Scalar forward substitution** restricted by conditional jumps. Actually done in source code).
- **Dead code elimination.**
- **Expansion** of scalars into arrays - equivalent of renaming inside loops. The idea is to remove dependencies between iterations.
- **Forward substitution** - restricted in scope by conditional jumps. Done (in Parafrase) to enhance tree height reduction.

I.4 Intermediate

(Used for Virtual Memory Machines (single instruction, multiple execution)):

- **Conditional Jumps**: either - sequential , or in parallel - for loops : compute mode vector and use it to mask stats in the scope of the conditionals. ("compress" only array refs that will be operated upon.)
- **Pattern matching** to recognize trivial test sequences replaceable by function calls (eg. max).
- **If removal** - boolean recurrence translation to linear system,and solving it. - Presumably at run time, since in general such a system can't be solved at compile time (more about this later).

I.5 Backend

(Used Mostly for Pipelined Processors):

- **Node splitting** in loops - Removes cycles in dependency by introducing intermediary stats.
- **Loop interchanging** to try to avoid recurrences in the inner loop.
- **Loop Distribution** (forms a separate loop from each strongly connected component). Good for generating array-ops and memory management.
- **Loop Fusion** - if distribution doesn't help, maybe fusion will. (mostly memory management).
- **Loop blocking** (for strip mining). Create vector operations of the size of the vector registers of the machine.

- **Removing invariants.**

I.6 Data-Dependence Analysis

The system differentiates between two types of analysis:

- Local flow => data-dependence testing ("conceptually equivalent to def-use chains").
- Global flow => pi-partitioning.

Since they believe that global flow analysis is inadequate because data flow ignores flow of control information and array indexes often depend on loop index and constants only, they rely solely on pi-partitioning for global flow information. This works as follows:

- Build *complete* dependency graph.
- Do pi-partitioning: A **pi-block** is a maximal set of statements cyclically connected by dependencies (possibly single statement). A recurrence is a cycle of data-dependency.
- Partitioning algorithm:
  1. Find pi-blocks.
  2. Partition into maximal anti-chains of pi-blocks (a set such that no 2 pi-blocks in it depend on each other). This partitioning is to preserve the relative order of the **dependent** blocks.
  3. Execute (schedule for execution) all members of the sets obtained in 2 above, in parallel. To enhance the number of things executable in parallel, the various techniques (source-to-source and graph transformations) are used.
- Apply graph abstraction (the idea is to isolate sets of statements which can be optimized (e.g. turned into array-ops) when taken as an ensemble. These are the transformations mentioned above.

I.7 Methods Used for Reducing Dependencies

To increase the exploitable parallelism the methods enumerated above are used. Their disambiguation system is integrated into this context, being used in building the dependence graph, and has an array-op orientation, as will be seen shortly. The following mechanisms are used in the process of building the *complete* dependency graph for the program:

I.8 Tree Height Reduction

Tree height reduction for arithmetic expressions, which allows their evaluation in O(log n) steps. This is enhanced by the use of back-substitution.

I.9 Recurrence Systems

A special mechanism deals with linear recurrences which can be expressed as:

$\bar{X} = \bar{C} + A\,\bar{X}$, with A being lower triangular
(i.e. x_i can't be a function of x_j, for j>i).

This is done by a preprocessor which looks at loops and classifies them as being of a certain type.(R<n,m> where n is the degree, and m is the bandwidth).

Next the system attempts to break recurrences into simpler ones:

Given a R<n,m> system:

Xk= funct(Xk-m1,Xk-m2,...,Xk-mr), where **m1<m2<..<mr=m**

It can be broken into g=gcd(m1,...mr) independent systems each of degree floor(n/g) or ceiling(n/g) and bandwidth: m/g.

To actually speed up execution times of recurrences which the system can handle, one of the following methods is used:

Column Sweep Algorithm:

- Broadcast known x_i's to every other equation (initially only x_0 is known)
- Multiply by corresponding coefficients (in parallel).
- Add in parallel to each partial sum.
- A new term becomes known: repeat, broadcasting the new x_i.

This takes O(n) steps given O(n) processors. Works on dependency-acyclic recurrences.

Product Form Recurrence Method:

Expand all x's as functions of the constant coefficients (a_{ij} and c_i), and then evaluate all the x_i's in parallel. (This is only true when x_i can't be a funct of x_j, for j>i - hence their assumption). This can be executed in O(2log n) steps, but becomes inefficient (requires huge numbers of processors) for effective speed-ups for higher order recurrences.

Wavefront Method:

The purpose of this method is to extract array-ops from higher order recurrences. Its generalization for n-dimensional arrays, the Hyperplane Method is essentially equivalent to Lamport's generalized Coordinate Method [27]. In fact Lamport's method is a cleaner formalization of several previous, less general approaches to the same problem.

The simplest version, for 2 dimensional arrays works as follows:

```
Given a Loop of the form :
   Do i1 = 1,n
   Do i2 = 1,m
     S(i1,i2)


in which S(i1',i2') is dependent on S(i1,i2),

and assuming:
  (a) i1-i1'= d1 = constant,
      i2-i2'= d2 = constant,
```

A computation wavefront (a line $L(\alpha)$) can be defined such that S(*,*) is computed simultaneously for all points on $L(\alpha)$, while it travels from left to right on the grid formed by the plane defined by the loop indexes. The trick is to choose α so as to minimize time (by doing as large a number of operations as possible in parallel), subject to keeping the computation valid (preserving dependencies).

By purely geometrical analysis the above conditions are expressed as functions of d1,d2 and the sizes of the index sets.

Notes:

- For the hyperplane method to work condition (a) above must be satisfied.
- Special case handling is required for handling back-references between loop bodies: S(i1',i2') will be updated before it's (old) value is used. Thus a kind of expansion is done to store the old value until it is used.
- Problems may arise when the bounds of recurrences are not known and/or lots of conditionals occur in the recurrences (see below). This may lead to degradation of performance.
- The Wavefront method seems to handle the whole loop body as one statement, making (a) above harder to satisfy, and implying that there is no partial overlapping of loop bodies other than of those specifically selected by the wavefront as parallel.

I.10 IF Nodes in Dependency Graphs

An if node is defined as a group of stats in which there are "many IF and GOTO statements and few arithmetic operations" [23]. Since conditionals are not usually solvable at compile time, their presence hinders the above methods of exploiting parallelism and requires special treatment. The idea is to put such sections of code into a canonical form, which may be more amenable to tree height reduction, and have more available parallelism. The canonical form is:

1. Set of assignment stats all executable in parallel.
2. Set of boolean functions all executable in parallel.
3. Binary decision tree equivalent to execution paths through the program section.
4. A collection of sets of assignment statements, each set associated with a path in the binary decision tree above. All assignments in each set can be done in parallel.

The "done in parallel" above is due to renaming and back-substitution and has nothing to do with the **If** handling proper. This **If** manipulation is designed to take advantage of special hardware (for boolean expression and decision tree evaluation). Some code explosion will result from the inevitable duplication in 4, but it is claimed to be within acceptable bounds.

I.11 IF Statements In Loops

The problem: "data dependency can be changed at execution time by the existence of such conditional statements...In the worst case, we may be forced by not knowing about control flow to compile loops for serial execution, which in fact can be executed in a highly parallel way".

To attempt to overcome this,the system uses a modified loop distribution algorithm that allows IF's. This is heavily array-op oriented/runtime(hardware) oriented. They try to transform the program so as to allow operating on whole arrays, and selectively omit certain elements as indicated by mode (mask) vectors which are set as a result of (pre)evaluation of the if's.

Parafrase discerns between 3 types of IF's in loops:

- If condition is independent of loop (type A). Handled by taking it out of the loop, and having separate loops, one for each branch. This shouldn't happen very often though.
- If condition depends on loop index(es) only. Use mode bits to "prefix" the body stat. These bits are set by using the if's, and are used throughout the loop (don't change).
- If condition depends on loop index(es), and loop body. Handled by executing each control path for the full loop(s) index set (in parallel), then use the merged outputs of

each control path to set the mode vector (which "prefixes" the loop body).

Appendix II. The Memory Anti-Aliasing System

In this appendix the top levels of the memory anti-aliasing system can be found, together with a definition of the interface between the disambiguator and the rest of the system. The system itself was written in a locally modified version of Maclisp.

II.1 The Interface

Here is the interface to the disambiguator:

- **(START-TRACE)**
  This signals to the disambiguator that the code-generator is about to start processing the next trace. The individual operations of the trace are presented via the function below. At this point the disambiguator clears its memory of obsolete information pertaining to the old trace, and prepares for the new one.

  This is just a signal to the disambiguator and does not return any meaningful information. (PREDECESSORS SOURCE-OPERATION TRACE-DIRECTION PTR) Presents the next source operation from the trace to the disambiguator. PTR is meaningful only to the code-generator.The disambiguator simply stores it and later returns it to signify that this operation is a predecessor. TRACE-DIRECTION is meaningful only if SOURCE-OPERATION is a conditional jump, and tells which way the jump will go on the trace. The disambiguator has to know which branch is taken so as to obtain the correct live-dead information to be used in establishing preemptive conditional dependencies (see chapter 5).

  PREDECESSORS returns the list of all previous operations on the trace that might be data predecessors of this operation and why they are data predecessors. The result is a list of sublists, each sublist of the form:

  (PRED REASON SOURCE-OPERAND SOURCE-TYPE PRED-OPERAND PRED-TYPE) Where:

  - PRED is a code-generator pointer that describes some source operation that SOURCE-OPERATION data-depends on (a PTR from a previous operation).
  - REASON is one of 'CONDITIONAL-CONFLICT, 'OPERAND-CONFLICT, or 'POSSIBLE-OPERAND-CONFLICT:

- CONDITIONAL-CONFLICT is returned for cases where PRED is a conditional jump above SOURCE-OPERATION (in source order) and SOURCE-OPERATION might write a location that is live at the top of the other leg of the jump.
- OPERAND-CONFLICT is returned when it is known that an operand of SOURCE-OPERATION is exactly the same as an operand of PRED, and that the use of the operands conflict (read after write, write after read, write after write).
- POSSIBLE-OPERAND-CONFLICT is returned when it is known only that an operand of SOURCE-OPERATON might be the same as an operand of PRED, and that the use of the operands might then conflict (read after write, write after read, write after write).

PRED-OPERAND and SOURCE-OPERAND are numbers that identify the conflicting operands of the corresponding source operations (first operand, second operand, etc.). Operands are numbered from left to right starting at 1. In the case of CONDITIONAL-CONFLICT, PRED-OPERAND will be ().

- SOURCE-TYPE and PRED-TYPE are either 'READ, 'WRITTEN, or 'CONDITIONAL-READ, and specify whether that operand was read, written, or read on the off-trace edge of a conditional jump.

Note that the result may contain several references to PRED, as the same operation may create multiple dependencies.

II.2 Block Definition and Flow-Graph Builder

```
;;;  definition of a block structure.
;;; ====================================================================
;;;

(def-struct stat-block

          fb.name         ;;; The block number.  That's its only unique name.

          stats           ;;; The source naddr code.

          conts           ;;; Continuations is a list of the form:(u) or (t f)
                          ;;; where u t f are:

          Uncond-cont     ;;; Uncond-cont -The next op,if this is an uncondjmp.

          True-cont       ;;; The next ops, if this is a condjmp.
          False-cont


          live-vars       ;;; Variables live entering this op.
```

```
        in              ;;; A list of the naddr stats reaching this one.

        change          ;;; T when IN has changed

        pred            ;;; In case this is the first block of a basic
                        ;;; block, all of the ops which directly
                        ;;; precede it.

        leader          ;;; signals the start of a basic block if true

        gen             ;;;  list of defs generated by the basic block.

        kill            ;;;  list of vars killed by the basic block.

        bb-ptr          ;;; pointer to coresponding bb.

           )


;;; (find-blocks)
;;; ========================================================================
;;; Finds blocks in the program and returns a list of blocks via the global *b*
;;; *B* is reversed at the end.
(defun find-blocks ()

  (let (  ( block-count   0   )      ( tmp       *prg* )
          ( instr         nil )      ( goto-lst  nil   )   )

    (options-find-blocks-in)

    (find-labels)  ;;; identifies jump targets and indexes them for easy access

    (fb  *prg*)     ;;; builds the flow graph of the program, using the structure
                    ;;; "block" defined in structs.lsp


    (:= *b*  (nreverse *b*))

    (set-conts)    ;;; sets the cont field for every stat-block


    ))



(defun fb (prg)

   (let (   (tmp    nil )   (tmp-sav nil )
            (tmp1   nil )   (instr   (skip-labels) )  )
   (if (:= tmp1 (hash-instr instr nil)) (then
      tmp1 )                                ;;; return
   (else
    (if prg (then (prog ()
      (:= tmp  (build-new-block instr))       ;;; new block w/ instr as stat
```

```
      (:= tmp-sav tmp)

;;; The next instruction is determined according to the type of the current one.
;;; for efficiency, sequential followers are procesed iteratively (and hence the
;;; "LBL"), whereas jumps are processed recursively.
lbl  (if (:= tmp1 (hash-instr instr nil) )  (then
      tmp1 )                                  ;;; return
     (else

      (selectq  (oper:group instr)

        ('if-then-else

            (insert-in-htable      )
                                               ;;; the test-succeeds label
            (process-if-true-label )

            (process-if-false-label)   )       ;;; (if given)

        ('cond-jump

            (insert-in-htable       )

            (process-cond-true-label )

            (process-cond-false-label )
                                              )

      ('goto
            (error 'shouldntbereached) )

      ('label
            (process-label-cont) )

      ('nil
            (error "unknown op-type - fb:" instr) )

      (t
            (add-array-def)   ;;; checks decls for array defs and adds them to lst

            (insert-in-htable)

            (process-uncond-cont) )
       )

   (return tmp-sav )          ;;; return CURRENT block
   ) ) ) ) ) ) ) )
```

II.3 The Interface Implementors

In this section we present the routines implementing the interaction with the outside world, as described by the interface definition above:

```
(defun new-programl  (*prg*)

    (let ( (sref-t    nil)  (matrix  nil)  (not-loop-flg nil)
           (sref-vars nil)  (n       nil)  (loop-starts  nil)
           (vars      nil)  (tmp     nil)  (loops        nil)
           (*dfn*     nil)  (leaders nil)  (init-vals    nil)
           (idx-lst   nil)  (lups    nil)  (ctr-loops      0)
           (straight  nil)  (*b*     nil)                    )

    (initialize.dis)

    (find-blocks)        ;;; Find blocks and sets  *B*  to the resulting graph.
                         ;;; The original seq of instructions is in  *PRG*.

    (dfn-order)          ;;; Find a dfs ordering of the blocks.
                         ;;; Resets *B* to this new list, and sets *DFN* to the
                         ;;; array of df numbers.

    (find-basic-blocks)  ;;; Find basic blocks starts
                         ;;; Sets  LEADERS,  the set of basic block startings ops.
                         ;;; and sets flags for them in *B*.

    (reach)              ;;; Find reach defs for basic blocks, also uses *B*  and
                         ;;; and leaders. Sets IN slots in *B* for LEADERS and
                         ;;; LOOP-STARTS.

    (const-fold)         ;;; When only one definition reaches a use, and it is
                         ;;; equal to a constant expression, the use is changed
                         ;;; to that value.  Then that is propagated, etc.
                         ;;; Uses *B* and LEADERS. Modifies stats in *B*

    (find-loops)         ;;; Finds the natural loops (see Dragon book).
                         ;;; Sets LOOP-STARTS to (more or less) the dominators.
                         ;;; Sets LUPS to a list of lists.  Each list on LUPs
                         ;;; is those ops not found in any contained loop.
                         ;;; Sets STRAIGHT to all the code not found in any loop,
                         ;;; and LOOPS to the leaders of the blocks in each loop.
                         ;;; Uses *B*, *DFN*, LEADERS.

    (process-seq-code)   ;;; Does constant and variable folding on the code
                         ;;; not in any loop.  This collects all expressions
                         ;;; a variable could be set to by the reaching defs.
                         ;;; It then folds the definitions as far as possible,
                         ;;; simplifying as it goes.  It throws away all
                         ;;; information no longer needed and produced for
                         ;;; this purpose.
                         ;;; Uses *B*, STRAIGHT, LEADERS, LOOPS, NOT-LOOP-FLG.
```

```
                     ;;; Modifies the reaching definitions in *B*.

    (process-lup-code)  ;;; Does CVF as above, but has to do more careful
                        ;;; work due to the painful presence of induction vars.
                        ;;; Uses  *B*,  STRAIGHT,  LEADERS,  LOOPS,  LUPS,
                        ;;; Modifies the reaching definitions in *B*.
                        ;;; Uses NOT-LOOP-FLG.

    (live-dead)         ;;; Sets up the live variables so that future calls to
                        ;;; predecessors will be able to return "jump precedence
                        ;;; edges".
                        ;;; Uses *B*, LEADERS, modifies the live field in *B*.

    (clear-globals.dis)

    ) )


;;;  PREDECESSORS finds dependencies  in a trace of instructions It will
;;;handle also the memory disambiguation.  The input  that  the  program
;;;expects is a stream of intermediate code instructions.
;;;
;;;  The format of the instructions that we assume  is :
;;;
;;; ( (instr# (instr-body))   (read regs list) (write regs list) (indx list) )
;;;
;;;  In the above,instr-body can be a list or an atom (so that it can be
;;;easily skipped - the dismbiguator needs  only  the  r/w  lists  ,  so
;;;that its only function is to make the i/o (somewhat) readable.
;;;  The program processes the instructions by  examining  them  one  at
;;;a time  and scheduling them in acordance with the precedences implied
;;;by the read/ write regs lists for each of them .  Once an instruction
;;;is done  it  is  no  longer needed (all that we do keep around is the
;;;last write ,and the reads access  for  each  register  encounterred).
;;;Renamming is  not  done,  by this program, so the last read reference
;;;is really useless.


(eval-when (compile)
    (require ellisp:vector)
    (require utilities:util)
    (require disambiguator:structs)
    (require interpreter:naddr) )

(include disambiguator:decls)

(declare (special -direction -pos -ptr -INST -CURR -ins *dis.conflict-type* ) )


;;;  RUNS THE WHOLE SHOW :( INIT I/O ); SET UP H-TABLE, UPDATES EACH INSTR;
(defun PREDECESSORS  (-ins -direction -ptr)
 (let ( (-<=   (copy  '(<=   )))
        (-<    (copy  '(<    )))   )
```

```
   (:= -instr (my-filter -ins)) ;;; Converts instr to the form:
                                ;;; (-ins (read-list) (write-list) (index-list))

   (update -INSTR  -ptr)        ;;; Updates read/written information in the regs
                                ;;; table (used for establishing the < and <=
                                ;;; lists) and resets <= and < to include any
                                ;;; dependency conflicts found.

   (update-cond-jumps )         ;;; Update-Cond-Jumps: handles the special case
                                ;;; of live vars  on  the non-trace  path  of a
                                ;;; conditional jump. (if direction is 1 or 0).

   (clean-up-lists )            ;;; Returns (< <=) w/o redundant information.

   ) )


;;; updates read/written refs for all regs in current instr, to the level where
;;; scheduled;also, check array references (as much as possible)
(defun UPDATE (-i -ptr)
   (let ( (-inst (copy -i)) )
        (if -inst (then
            (update-rlist )
            (update-wlist ) ) ) ) )


;;; Update-Cond-Jumps: handles the special case of live vars on the alternate
;;; paths of a conditional jump (the non-trace path), in a way similar to UPDATE
;;; the let masks <=,<, lists so as not to consider the updates in
;;; current op (they shuold apply only for ops BELLOW jmp, not ABOVE)
(defun UPDATE-COND-JUMPS ()
   (let (  (-<=  (copy '(<=   )))
           (-<  (copy '(<     )))  )

      (if -direction  (then       ;;; add constraints for cond jumps

          (selectq -direction

             ( 'right             ;;; false is taken, get live from true branch
                (update-cjmps (b:l (b:t (hash-instr -ins nil))) -ptr))

             ( 'left              ;;; true is taken, get live from false branch
                (update-cjmps (b:l (b:f (hash-instr -ins nil))) -ptr))

             (t (error "illegal direction code (not right/left/nil)") )
             ) ) ) ) )


;;; updates cond-jumps lists for vars also, check array references
;;; (as much as possible)
(defun UPDATE-CJMPS (lst  -ptr)
   (LOOP (INITIAL   -TMP NIL   -CURR NIL   -RL lst -pos nil)
         (WHILE   (:= -CURR (POP -RL)) )
```

```
(DO (if (&& (listp -curr)  ;;; added
            (ASSOC (get-var-name -CURR ) -CURR-ARRAY-LST)) (then
        (let* ( (index       (cadr -curr))
                (-curr-array (assoc (get-var-name -curr)
                                    -curr-array-lst) )
                (ptr         (caddr  -curr))

                (tmp         nil)  )

         (if (:= tmp (assoc index (cdr -curr-array))) (then
             (:= (car (cddddr tmp))
                 (append1 (car (cddddr tmp)) '(,-ptr ,-pos))))
         (else
             (nconc1  -curr-array
                      (copy  '(,index ,ptr nil nil
                                            (,'(,-ptr ,-pos))))`) ) )
        ) )
                    .
    (else  ;;; add things only if not already there
     (if (! (member '(,-ptr ,-pos) (caddar (last (hash -curr))))) (then
         (push '(,-ptr ,-pos) (caddar (last (hash -curr)))) ) ) )
    ) ) ) )


;;; Updates read information in the table. Used in UPDATE.
(defun update-rlist ()
   (loop (initial
          -tmp   nil        -rl    (getrlst -inst)
          -curr  nil         r     nil
           w     nil         cj    nil)
        (while
            (desetq (-curr -pos)  (pop -rl)))
        (do
            (if (array? -curr) (then
                (desetq (r w cj)
                        (check-and-update-array  'read  -curr  -instr  -ptr) )
                (add-last-written-array-to-<-preds read ) )
             (else
                (add-last-written-to-<-preds read)
                (update-reads-list)
                  ) ) ) ) )

;;; Updates writen information in the table. Used in UPDATE.
(defun update-wlist ()
   (loop (initial
          -tmp  nil        -curr nil
           r    nil        -wl   (getwlst -inst)
           w    nil         cj   nil              )
        (while
            (desetq (-curr -pos) (pop -wl)) )
        (do
            (if (array? -curr) (then
                (desetq (r w cj)
```

```
                (check-and-update-array 'write -curr -instr -ptr) )
            (add-cond-jumps-array-info-to-<-preds written )
            (add-read-array-info-to-<=-preds written)
            (add-last-written-array-to-<-preds written ) )
        (else    ;;; save w/o duplicates , and don't include current one
            (add-cond-jumps-info-to-<-preds written )
            (add-read-info-to-<=-preds written)
            (add-last-written-to-<-preds written)
            (update-and-reset-written-reg-info)))
             ) ) )
```

```
;;; Cleans < and <= lists of any redundant information and returns them in the
;;; form :    ( <  <= )
(defun CLEAN-UP-LISTS ()
   (cleanup -<  -<=  -ptr )) ;;; Removes redundant information from  list.


;;; resets vars needed and (also def in new.lsp), and clears hash table for last
;;; read/written
(defun START-TRACE ()

   (:=  -instr nil)
   (:=  H-ARRAY (MAKE-VECTOR 31))

;;; reset array info specific to trace
   (for (i in -curr-array-lst) (do (:= (cdr i) nil)))
   nil)
```

II.4 Memory Reference Comparator

This section contains the overall disambiguation mechanism, i.e. the part that actually accepts two index expressions and decides how best to compare them.

```
;;; Given a b c in ax + by = c, and (optionally) either  values  of x, y
;;; or bounds for x,y. returns nil if no collisions can occur, or a list
;;; of collision pts otherwise (i.e. ((x-colission1 y-colission1) .....)  )
;;; loopflg is a variable that can have the following vals/meanings:
;;;         - 0 / no expr is from inside a loop.
;;;         - 1 / first expr is from inside a loop, the second is not
;;;         - 2 / second expr is from inside a loop, the first is not
;;;         - 3 / both exprs are from inside a loop
;;; bounds-vars is a list of the form: ((varname (list of vals))...)
;;; if the list of vals comes from a doloop with step =1 the first element
;;; should be "step".
```

```
(defun  generate-solution (expr1 expr2  x y  bounds-vars)
 (:= *dis.conflict-type* 'possible-operand-conflict )
 (if (&& *full-disambiguation* expr1 expr2) (then
  (let ((x1 nil)(y1 nil)(a nil) (b nil) (c nil) (g nil) (b-x nil) (b-y nil)
        (*vars* nil)  (expr '(nadd ,expr1 (nmul -1 ,expr2))))
       (:= expr  (simplify expr))
       (? ((my-zerop expr) ;;; identical expressions (or identically valued)
           (:= *dis.conflict-type* 'operand-conflict )
           t)  ;;; always collide
          ((numberp expr)
           nil)   ;;; never colide
          ((&& (numbersp  expr1)     ;;; lists of numbers
               (numbersp  expr2))    ;;; note that if only one is numbers,it is
                                     ;;; better to treat them both as vars, and
                                     ;;; the vals as bounds-var
           (intersect (numbersp expr1)(numbersp expr2)))
          ((&& (listp expr) (! (nop:is expr))) ;;;i.e. a list
           (for (i in expr) (filter ;;; either all nil, or return colisions
                (generate-solution i 0 x y bounds-vars))))
          ((progn (desetq (a b c g) (check-expr expr bounds-vars))
                  (&& (numberp a) (numberp b) (numberp c)))
           ;;; check-expr also resets expr to have <= 2 vars if needed ;
           ;;; if less than 2 vars are found, but a long list
           ;;; is found in expr, check-expr creates a dummy var, introduces
           ;;; its vals as bounds in bound-vars and if still <= 2 vars are
           ;;; present, we can solve dioph. eq. (else nil is returned)
           (desetq (x1 y1) (solve-di2 a b c))
           (if x1 (then        ;;; if solutions for x exist
               ;;; check bounds list for conforming to solution (special cases)
            (if(:= b-x (cadr (assoc x bounds-vars)))(then ;;; there is a b list
               (if (&& (== (car b-x) 'step)  ;;; monotonously ascending sequence
                       (pop b-x)             ;;; get rid of flag
                       (|| (&& (> b 0) ;;; x is ascending
                               (> x1 (car (last b-x))))
                           (&& (< b 0) ;;; x is descending
                               (< x1 (car  b-x))))) (then
                     (:= b-x nil));;; no collisions can occur (no t can satisfy)
                                  ;;; in x = x1 + b/g * t
                   (else
                    (:= b-x
                       (for (x in (copy b-x))(filter ;;; get t vals for x
                            (let ((tmp (// (-  x x1) (// b g))))
                              (if (&& (= (rem (-  x x1) (// b g)) 0)
                                      (>= tmp 0)) (then
                                   tmp)
                              (else nil)))))))))
            (else (:= b-x t)))))
               ;;; check bounds list for conforming to solution (special cases)
           (if y1 (then        ;;; if solutions for x exist
               ;;; special case for unifirmly increasing interval(s):
            (if(:= b-y (cadr (assoc y bounds-vars)))(then ;;; there is a b list
               (if (&& (== (car b-y) 'step)  ;;; monotonously ascending sequence
                       (pop b-y)             ;;; get rid of flag
                       (|| (&& (< a 0) ;;; y is ascending
```

```
                        (> y1 (car (last b-y))))
                    (&& (> a 0) ;;; y is descending
                        (< y1 (car  b-y))))) (then
               (:= b-y nil)) ;;; no collisions can occur (no t can satisfy)
                    ;;; in y = y1 - a/g * t
             (else
              (:= b-y
                 (for (y in (copy b-y))(filter ;;; get t values for y
                      (let ((tmp (// (-  y y1) (// b g)))
                        (if (&& (= (rem (- y y1) (// b g)) 0)
                                (>= tmp 0)) (then
                            tmp)
                        (else nil))))))))))
      (else (:= b-y t))))) ;;; if no restrictions (bounds) but y1#nil => t
     (? ((&& b-x b-y)
        (intersect b-x b-y)) ;;;
        (b-x)
        (b-y)))
    (t t))))   ;;; safe way out => if can't say anything, assume collision
 (else t)))    ;;; one of expr1 or expr2 is nil, => no info,
               ;;; => possible colision
```

II.5 Solution to Diophantine Equation

The solution to the two variables diophantine equation is as follows: Given ax + by = c, find g=gcd(a,b). If the remainder of dividing c by g is not an integer, no collisions can occur. Else find x0,y0 s.t. g=ax0+by0, (Euclid) and get x1=(c/g)x0, y1=(c/g)y0. other solutions (i.e. interference points between the refs.) are:

xi= x1 + b*i, yi= y1 - a*i, For all i in Z.

This is essentially the traditional solution for diophantine equations in two unknowns to be found in any Number Theory book. The only trick in implementing it is to find x0 and y0. This is easily done by running Euclid's algorithm for GCD backwards. The implemenation details are given below.

```
;;; remainder macro, which avoids bug in compiled \.
(defmacro rem (x y)
  `(if (= ,y 0) (then ,x)
   (else (\ ,x ,y))))


(defun gcd1 (b c)
    (let* ((remainders nil) (quotients nil) (qi 0)(qi-1 0)(tmp 0)
```

```
         (ri-1 (rem b c)) (ri (rem c ri-1)))
        (push ri-1 remainders)     ;;; r1
        (push ri remainders)       ;;; r2
        (push (:= qi-1 (// b c))  quotients)   ;;; q1
        (push (:= qi   (// c ri-1)) quotients)   ;;; q2

       (loop (while (! (= ri 0)))
             (do
              (push (//  ri-1 ri) quotients)
              (:= tmp ri)
              (push (:= ri (rem ri-1 ri)) remainders)
              (:= ri-1 tmp))
             (result (list (progn (pop remainders) (car remainders))
                           remainders quotients)))))

;;;
;;; finds coefficients  of b,c in b*x0 + c*y0 = gcd(b,c)
;;;
(defun findcoeff   (b c)
    (let ((remainders nil) (quotients nil) (gcd nil) (qc nil) )
         (desetq (gcd remainders quotients) (gcd1 b c))
         (:= qc         ;;; coefficient of c
             (fc1 (reverse remainders) (reverse quotients)
                        (length remainders) 0) )
         (list  ;;; return coeff of b and c as a list
             (// (- (gcd b c) (* qc c)) b) qc)))


(defun fc1 (r q j n)
  (? ((= n (1- j)) (- (nth (1- j) q)))
     ((= n (- j 2)) (+ 1 (* (nth (1- j) q) (nth (- j 2) q))) )
     (t (- (fc1 r q j (+ 2 n)) (* (fc1 r q j (1+ n)) (nth     n  q))))))


;;;
;;; solves diophantine eq. in two unknowns of the form: ax + by = c
;;;
(defun solve-di2 (a b c)
    (let ((x0 nil) (y0 nil)(x1 nil) (y1 nil)(g nil))
         (:= g (gcd (abs (fixnum-identity a)) (abs (fixnum-identity b)) ) )
         (if (!= (rem c g) 0) (then
            nil) ;;; no collisions
         (else
          (desetq (x0 y0) (findcoeff (abs (fixnum-identity a))
                                     (abs (fixnum-identity b))))
          (:= x1 (* (// c g) x0))
          (:= y1 (* (// c g) y0))
          ;;; compensate for lost info (from  using abs value)
          (? ( (= 0 (- (+ (* x1 a) (* y1 b)) c))          '(,x1 ,y1))
             ( (= 0 (- (+ (* (- x1) a) (* y1 b)) c))      '(,(- x1) ,y1))
             ( (= 0 (- (+ (* x1 a) (* (- y1) b)) c))      '(,x1 ,(- y1)))
             ( (= 0 (- (+ (* (- x1) a) (* b (- y1))) c))  '(,(- x1) ,(- y1)))
             (t (error "impossible")))))))
```

II.6 Generalized Constant and Variable Folding

```
;;;  (process-seq-code)
;;; ========================================================================
;;; Does cvf on the sequential part of the code (i.e. the code not in any loop)
;;; This incrementally collects all expressions a variable could  be set to by
;;; the reaching defs.  It then  folds  the  definitions  as  far  as possible,
;;; simplifying as it  goes.  It throws  away all information no longer needed
;;; and produced for this purpose.
;;; Uses *B*, STRAIGHT, LEADERS, LOOPS, NOT-LOOP-FLG.
;;; Modifies the reaching definitions in *B*.
(defun process-seq-code ()
   (if *full-disambiguation* (then
       (process-seq-code1) ) ) )

(defun process-seq-code1 ()

  (let (  (tmp nil)  (vars nil)  (sref-vars nil)  )

       (options-process-seq-code-in)

       (get-srefs-seq)

       (filter-read-vars)

       (:= not-loop-flg  t ) ;;; this is not a loop - take this into accout
                             ;;; doing the cv folding

       (:= tmp leaders)      ;;; all leaders in straight code
       (for (l in loops) (do (:= tmp (set-diffq tmp l)))) ;;; prog - loops (LEADERS)

       (for (b in tmp) (do
            (let ( (blk  b)
                   (in  (b:i b)) )
                 (:= (b:i b) nil)
                 (cvf-basic-block 'const-var-fold-seq-stat blk) ) ) )

       (clean-up.cvf-seq)

       ) )


;;;  (process-lup-code)
;;; ========================================================================
;;; Does CVF as above, but has to do more careful work due to the painful
;;; presence of induction vars. In particular, the order in which assignments to
;;; induction vars appear in the loop body, relative to their uses is critical.
;;; Uses  *B*,  STRAIGHT,  LEADERS,  LOOPS,  LUPS, NOT-LOOP-FLG.
;;; Modifies the reaching definitions in *B*.
```

```
(defun process-lup-code ()
   (if *full-disambiguation* (then
       (process-lup-code1) ) ) )


(defun  process-lup-code1 ()

  (let ( (sref-vars nil)  (vars nil) )

       (options-process-lup-code-in)

       (:= not-loop-flg nil) ;;; this IS a loop - treat it as such in cvf-loop

       (initialize.lup)      ;;; set-up loop processing

       (for (lup in loops) ;;; lup is used for the real thing, (bblocks)
            (l   in lups ) (do
            (let ( (tmp nil) )

            (for (b in lup) (do
                 (let ( (blk b)
                        (in (b:i b)) )
                        (:= (b:i b) nil)
                        (cvf-basic-block 'const-var-fold-lup-stat blk)))) ) ) )

       (clean-up.cvf-lup)
       ) )


 (defun cvf-basic-block  (cvf-fun blk)
    (loop (while blk)
          (do   (funcall cvf-fun blk))
          (next  blk (b:u blk) )
          (until (|| (! blk) (leader? blk)) ) ) )


;;;=================================================================
;;; Puts the reach defs (in out) into a more space efficient form.
;;; ((varname def1 def2 ... defn))where defi is one of: #, var, (op * * )
;;; init-vals is of the form: ((varname initial value)....) this information is
;;; to come from the meta information  found in loop definitions.

(defun convert-reach-defs (i sref-vars init-vals )
  (let (  (tmp nil)  (tmp1 nil)  (stat nil)  (tmpin nil)  )
      (:= stat  (b:s i))
    ; (:= srefs (if mx (then (union sref-vars
    ;                               (find-trans-refs stat vars mx n)))
    ;           (else sref-vars)))
      (:= tmpin  nil)
      (for (j  in  (copy (b:i i)) )
           (when   (defs-read-vars:cvf  (oper:dest j)  stat  sref-vars) )
           (do
             (standardize-ops j)
             (group-ops       j) ) )
```

```
      (:= (b:i i)  (nreverse tmpin) ) ) )




(defun const-var-fold-seq-stat (blk)
   (if (read-ops? blk) (then
       (set-in)
       (convert-reach-defs blk vars  () )
       (cvf-loop  blk)        ;;; keeps only reach defs for vars read
                              ;;; by stat and those trans   reachable
       (discard-useless-IN-info) )
   (else
     (set-in)
     (if (!== (oper:group (b:s blk)) 'loop-start) (then
         (:= (b:i blk) nil))))) )


;;; the difference between the 2 cvf routines (seq/lup) is in the handling of
;;; recurences
(defun const-var-fold-lup-stat (blk)
   (if (read-ops? blk) (then

       (set-in)
       (convert-reach-defs blk vars () )
       (:= (b:i blk)  (fix-srefs vars (b:i blk))) ;;; order stats
       (cvf-loop  blk)        ;;; keeps only reach defs for vars read
                              ;;; by stat and those trans   reachable
       (discard-useless-IN-info) )

   (else

     (set-in)
     (:= (b:i blk) nil)) ) )


;;; (cvf-loop bl)
;;;======================================================================
;;; does the generalized const folding var folding ; it works only on the output
;;; of (convert-reach-defs bl) and does not touch the original statements
;;; (unlike real cf, there is no real transformation to the program).

;;; the cv folding is done incrementally:
;;; we start with the vars read in the current statement, and try to fold them;
;;; in so doing, we "touch some other vars, which are added to a list of vars
;;; touched (to be folded. the process stops when there are no changes and no
;;; new vars have been added to the touched-list
(defun cvf-loop (i)
    (let ( (change t)    (*ctr-lps* 0)
           (touched-lst  (for (k in (b:i i)) (filter
                              (if (read-indexes (car k) (b:s i)) (then k)))))  )

        (loop (while change) (do
```

```
            (:= change nil)
            (for (k in (fix-srefs vars touched-lst)) (do
                 (++ *ctr-lps*)
                 (:=  *current-top*  k)
                 (mapc  (f:l (x) (if x (:= change t)) )
                        (maplist 'cvf  (cdr k) ) ) ) ) ) )
     ) )
```

II.7 Loop Finder

```
;;; (find-loops)
;;; ========================================================================
;;; Finds the natural loops (see Dragon book). Sets LOOP-STARTS to (more or
;;; less) the dominators.  Sets LUPS to a list of lists.  Each list on LUPs
;;; is those ops not found in any contained loop. Sets STRAIGHT to all the
;;; code not found in any loop, and LOOPS to the leaders of the blocks in each
;;; loop. Uses *B*, *DFN*, LEADERS.
;;; Assumes predecessors are left in place by a function like reach
(defun find-loops ()
   (if *full-disambiguation* (then
       (find-loops1) ) ) )


(defun find-loops1()
   (let* ( (len  (add1 (length *b*)))   (backedges  (find-backedges *b*))
           (*stack-1*  nil)             (*stack*    nil)
           (*loop*     nil)             (tmp        nil)
           (i          (- len 1))       (*lups*     nil)  )

        (options-find-loops-in)

        (initialize.find-lup)

        (find-predecessors)       ;;; find  predecessors  for each statement

        (find-mixed-loops )       ;;; find the loops w/o separating them

        (get-loop-starts)

        (clean-up.find-lup)       ;;; save space

        (separate-loops)          ;;; break appart nested loops

        (get-loop-leaders)        ;;; starting basic blocks for each loop

        (get-straight-code)       ;;; get all stats in non-loop code

        ) )
```

```
;;; (dfn-order)
;;; ==========================================================================
;;; Rearanges b-list in dfs order  and resets *B* to this new list
;;; Also sets the array of df numbers, *DFN*.


(defun dfn-order  ()
  (let* (
     ( len         (add1 (length *b*)) )
     ( *marked*    (make-bits len)     )
     ( *stack-1*   nil                 )
     ( *stack*     nil                 )
     ( *loop*      nil                 )
     ( i           (- len 1)           )
     ( *lups*      nil                 ) )

    (options-dfn-order-in)

    (:= *dfn*  (array () fixnum len))

    (dfs (car *b*))

    (:= *b*  (sort *b* 'dfn-sorting-function) )

    ) )


;;; finds backedges using df-numbering information
(defun find-backedges (b-list)
   (let ( (backedges nil) )
        (for (m in b-list) (do
             (for (n in (b:ct m))
                  (when (>= (arraycall fixnum *dfn* (b:n m))
                            (arraycall fixnum *dfn* (b:n n)) ) )
                  (do (push '(,m ,n) backedges) ) ) ) )
        backedges ) )



;;; prepare for find-lups - clear predecessors
(defun initialize.find-lup ()
   (for (b in *b*) (do
        (:= (b:p b)  nil)) ) )


;;; finds predecessors for each statement
(defun find-predecessors ()
        (for (b in *b*) (do
             (for (s in (b:ct b)) (do
                  (push b (b:p s))))) )
```

```
;;; find the loops w/o breaking them apart
(defun find-mixed-loops ()
   (for ((n d)  in  backedges) (do
        (:= *stack* nil)
        (:= *loop* '(,d))
        (insert  n)
        (:= *lups* (append1 *lups*  (find-loop) ) ) ) ) )


;;; find the first stats in a loop (loop-start's)
(defun get-loop-starts ()
   (:= loop-starts
       (for (i in *lups*) (splice
            (for (j in (b:p (car (last i))))
                 (when (== (oper:group (b:s j)) 'loop-start))
                 (save j))))) )


;;; finds one single loop given its start. (this loop may include other loops'
;;; stats if it is an outer loop.
(defun find-loop ()
   (loop (initial m nil)
         (while *stack*)
         (do
           (:= m (pop *stack*))
           (for (p in (b:p m)) (do (insert p))))
         (result *loop*)))
```

II.8 Simplifier/Canonizer

```
(defun ordr (x y)
    (? ((&& (listp x) (listp y)) (> (length x) (length y)))
       ((listp x) (if (numberp y) (then t)
                  (else (if (atom (car x)) (then
                             (alphalessp (car x) y))
                        (else t)))))
       ((listp y) (if (numberp x) (then nil)
                  (else (if (atom (car y)) (then
                             (alphalessp x (car y)))
                        (else nil)))))
       ((&& (numberp x) (numberp y)) (> x y))
       ((numberp x) nil)
       ((numberp y) t)
       (t (alphalessp x y))))

(defun simplify (x)
    (? ((&& (listp x) (! (assoc (car x) -curr-array-lst)))
        ;;; either operation or list of vars
```

```
        (if (nop:is x) (then
            (nop:apply (opr:nop x)
                       (for (i in (operands:nop x)) (splice
                            (no-dupls-op (opr:nop x) (simplify i))))))
        (else (if (operation:is x) (then x) ;;; do nothing, just leave it alone
              (else
               (for (i in x) (splice (no-dupls-op (opr:nop x) (simplify i)))))))
           ))
       (t x )))      ;;; variable or number or array


(defun no-dupls-op (opr x)
   (? ((&& (listp x) (nop:is x) (== (opr:nop x) opr));;;if we have (op ..(op..))
       (operands:nop x))
      (t '(,x))))



(defun nop:apply (opr l)
  (let ((u nil) (r (if (== 'nadd opr) (then 0) (else 1))) (lst nil) (rl 0)
        (s nil))
       (:= rl r)
       (for (i in l) (do (? ((numberp i) (:= r (funcall opr r i)))
                            ((&& (listp i) (nop:is i)) (push i u))
                            ((&& (listp i) (! (assoc (car i) -curr-array-lst)))
                             (push i lst))
                            (t (push i u)))))
    (:= lst (loop (initial l1 (pop lst) l2 (pop lst) tmp nil)
                  (while l2)
                  (do
               (:= tmp (for (i in l1) (splice (for (j in l2) (save
                                             (simplify '(,opr ,i ,j)))))))
               (:= l1 tmp)
               (:= l2 (pop lst)))
             (result l1)))
    (:= lst (no-dupls lst)) ;;; at this point we have only one list

    (? ( lst
        (? ((&& (== 'nadd opr) (=  r 0)) (:= lst (sort lst 'ordr)))
           ((&& (== 'nmul opr) (=  r 0)) (:= lst nil))
           ((&& (== 'nmul opr) (=  r 1)) (:= lst (sort lst 'ordr)))
           ((my-zerop lst) (if (== 'nmul opr) (then (:= r 0)) ;; so result is 0
                         (else (:= lst nil)))) ;; so we don't print "0"
           (t (:= lst (no-dupls (for (i in lst) (splice
                                '(,(simplify '(,opr ,i ,r)))))))
              (:= lst (sort lst 'ordr))))))

(:= s (? ( u
         (? ((&& (== 'nadd opr) (=  r 0) lst)  ;;; distribute over lst
              (:= u  (sort u 'ordr))
              (no-dupls (for (i in lst) (splice
                             '(,(simplify '(,opr ,i ,@u))))))
           ((&& (== 'nadd opr) (=  r 0))
```

```
          (if (> (length (:= u (sort u 'ordr))) 1) (then
              '(,opr ,@u))
          (else (car u))))
         ((&& (== 'nmul opr) (=  r 0)) 0)
         ((&& (== 'nmul opr) (=  r 1) lst)
           (no-dupls (for (i in lst) (splice  (for (j in
                                                  (:= u (sort u 'ordr)))
                         (splice '(,(simplify '(,opr ,i ,j)))))))))
         ((&& (== 'nmul opr) (=  r 1))
          (if (> (length (:= u (sort u 'ordr))) 1) (then
              '(,opr ,@u))
          (else (car u))))
         (lst
           (no-dupls (for (i in lst) (splice  (for (j in (sort u 'ordr))
                         (splice '(,(simplify '(,opr ,i ,j)))))))))
         (t '(,opr ,@(sort u 'ordr) ,r))))
       (t  (if lst (then lst)
           (else r)))))
(canonize s))


(defun canonize (s)
 (let ((add1 nil) (adds nil) (muls nil)(mults nil))

  (? ((&& (listp s) (== (car s) 'nmul)(assoc 'nadd s))
                                    ;;; s is of the form: (* .. (+ ..)..)
      (for (i in (operands:nop s)) (do
           (if (&& (listp i) (== (car i) 'nadd)) (then
               (push i adds))
           (else (push i muls)))))
      (:= add1 (pop adds))
      (let ((m (pop muls))) ;;; transform to mults using first add :
                                    ;;; (* a b (+ c d)..) => (+ (* c b a)(*.))
         (:= mults (for (i in (operands:nop add1)) (save '(nmul ,i ,m))))
         (:= m (pop muls)) ;;; mults is a list of the form ((* x x ) ...)
         (loop (while m)
               (do
                  (:= mults (for (i in mults) (save (append1 i m)))))
               (next m (pop muls))))
      (:= mults
          (?  ( (for (i in adds) (splice (for (k in (operands:nop i)) (splice
                    (for (j in mults) (save (append1 j k))))))) )
              (t mults)))
      (simplify (append '(nadd) mults)))
    ((&& (listp s) (== (car s) 'nadd)(assoc 'nmul s))
                        ;;; where s is of the from (+ (* . .) ..(* ..)..)
      (let (( terms nil)(others nil) (tmp nil)(add-terms nil)(change nil))
                (:= add-terms (cdr s))
        (for (i in add-terms) (do ;;; transform (+ (* a .)(* a .) ) to
                                ;;; (+ (* a (+ (* .) (* .)))),in hope of smpl
            (if (|| (&& (listp i) (== (opr:nop i) 'nmul))
                    (&& (listp i) (memq (car i) -curr-array-lst)
                                  (:= i (copy '(nmul ,i 1))))
```

```
                    (&& (! (numberp i)) (:= i (copy '(nmul ,i 1))))) (then
                (? ((:= tmp (cadr(assoc (car (operands:nop i)) terms)))
                                        ;tmp=nop
                    (:= change t)
                    (:= (cddr tmp )
                        (list (simplify
                                  '(nadd (nmul ,@(cddr tmp))
                                          (nmul ,@(cddr i)))))))
                   (t (push '(,(car (operands:nop i)) ,i) terms))))
            (else (push i others)))))
                        ;;; transform (+ (* a ..).a.) to (+ (* a (+ 1 ..))..)
    (:= others (for (i in others) (filter
            (if (:= tmp (cadr(assoc i terms))) (then
                (:= change t)
                (:= (cddr tmp)
                     (list (simplify '(nadd (nmul ,@(cddr tmp)) ,1))))
                nil) ;;; erase it from the others-list
            (else i)))))

      (:= add-terms (append (for (i in terms) (save (cadr i)))
                              others))
             (append '(nadd) add-terms)))
    ((&& (listp s) (== (car s) 'nadd)) ;;; just (+ ...)
     (let (( terms nil)(others nil) (tmp nil)(add-terms nil)(change nil))
      (:= add-terms (cdr s))
      (for (i in add-terms) (do
           (if (|| (&& (listp i) (memq (car i) -curr-array-lst)
                       (:= i (copy '(nmul ,i 1))))
                   (&& (! (numberp i)) (:= i (copy '(nmul ,i 1))))) (then
               (? ((:= tmp (cadr (assoc (car (operands:nop i)) terms)))
                   (:= (caddr tmp) (+ (caddr tmp) 1))
                   (:= change t))
                  (t (push '(,(cadr i) ,i) terms))))
           (else (push i others)))))
      (:= add-terms (append (for (i in terms) (save
                                  (simplify (cadr i))))
                              others))
      (if change (then (simplify  (append '(nadd) add-terms)))
      (else (append '(nadd) add-terms)))))

    (t s))))

(defun no-dupls (l)
  (let ((tmp nil))
   (for (i in l) (when (! (member i tmp))) (do (push i tmp)))
   tmp))

(defun nadd (x y)
 (plus x y))

(defun nmul (x y)
    (times x y))
```

Appendix III. The RunTime Disambiguation System

This file contains all the modifications to the bulldog compiler which are required for run time disambiguation. The files from which code was taken are tr:bookkeep.lsp tr:compact.lsp and exp:skex.lsp . First, the procedures which are MODIFIED are presented, and the files from which they originate are given. In each such procedure, the code added is enclosed in ;;;***begin-insert***;;; and ;;;***end-insert***;;; comments in the lines imediately preceding and succeding the inserted code. All teh rest of teh code in these procedures is part of the original trace scheduler, as written by J.Ellis and J. Fisher.

Functions which do not have a reference to an original file were defined and used exclusively for the run time disambiguator.

```
;;; This procedure comes from  tr:bookkeep.lsp
(defun bk.fix-one-flowsu ( inx )

(let ( (this-mi    (*coal* inx) )
       (jump-elts (bk.mi:cond-jump-constituents (*coal* inx) inx) ) )


    ;*** case 1:  we scheduled an instruction not containing a conditional
    ;*** jump.  In that case, its successor is simply the next scheduled
    ;*** instruction.  Make sure that we eject the proper DEFs after the
    ;*** last cycle of the schedule.

    (if (not jump-elts) (then
        (:= (mi:lstflowsu this-mi) (list (*coal*  (1+ inx) ) ) )

        (if (= inx *schedsize*) (then
            (bk.splice-def&partial-schedule
                this-mi
                (schedule:split *schedule* inx *all-vars*)
                (car (mi:lstflowsu this-mi) ) ) ) ) )

    ;*** case 2:  the instruction scheduled at inx has at least one jump among
    ;*** its constituent mops. Consider each cond jump in turn.

    (else

    ;*** The successors of the new MI are those of the conditional jumps, with
```

```
;*** the cond.  jumps' tracesus replaced by the next scheduled instruction.
;*** We update the off the trace flowprs to reflect this, but not those
;*** on the trace, since we're going to fix those right in the next pass,
;*** anyway.

    (loop (for jump-elt in jump-elts)
          (initial succ-list ()
                   succ      () )
          (next succ (bk.mi:off-tracesu jump-elt) )
    (do
        (push succ succ-list)
        (:= (mi:lstflowpr succ)
            (top-level-substq this-mi jump-elt
                              (mi:lstflowpr succ) ) ) )
    (result
        (push (*coal* (1+ inx) )
              succ-list)
        (:= (mi:lstflowsu this-mi)
            (nreverse succ-list) ) ) )

;*** Check to see if any elements were before the cond jump on the
;*** trace, but have now been scheduled below.  For each, call it
;*** ti, we make a copy of it for placement before each off-trace
;*** follower.  Note that these tis are formed in reverse trace order,
;*** necessary since data precedence must be preserved in the sequence
;*** of copied instructions.  Thus the "decr" in the loop below.  After
;*** the copies are made, we splice in the def and partial schedule
;*** of multi-cycle operations that spanned the split point that the
;*** code generator told us about.
;***
;*** Before splicing each split edge looks like:
;***
;*** this-mi -> dummy -> off-trace
;***
;*** After, it looks like:
;***
;*** this-mi -> partial-sched -> def -> split-copies -> dummy -> off-trace
;***
;*** (For now, we have an option to put all the compensation code on
;*** the other side of the dummy).
;***

    (let ( (partial-schedule&def
               (schedule:split *schedule* inx *all-vars*) ) )

        (loop (for jump-elt in jump-elts)
              (bind jumper this-mi
                    jumpee (bk.mi:off-tracesu jump-elt) )
        (do
            (loop (decr inx2 from (1- (mi:tracepos jump-elt) ) to 1)
                  (bind ti (*trarray* inx2) )
            (do
                (if (&& (mi:first-cycle ti)    ;*** scheduled?
                        (> (mi:first-cycle ti) inx)
```

```
                              ;;;***begin-insert***;;;
                              (remove-rt-disamb-code ti) ;;; remove if returns nil
                              ;;;***end-insert***;;;
                              )
                         (then
                            (:= jumpee (bk.splice (list jumper)
                                                  (bk.mi:copy ti 'split)
                                                  jumpee) )
                            (++ *split-count*)
                            (if *print-copying* (then
                               (bk.print-copy-after-message ti jump-elt) ) )))))

                (bk.splice-def&partial-schedule
                    jumper
                    partial-schedule&def
                    jumpee) ) ) ) ) )

   () ) )

;;; finds the on-trace vop using index idx, scheduled at cycle >= CYCLE.
;;; used in find-vops-pair below
(defun find-vop-mi(idx cycle)
  (car (for (i in *tr*)
            (when (&& (memq (oper:group (car (mi:source i)))
                            '(vload vstore))
                      (== idx (oper:part (car (mi:source i)) 'index))
                      (>= (mi:first-cycle i) cycle)))
        (save i))))

;;; Given a pair of indexes and a cycle returns 2 vops at or below cycle, that
;;; use idx1 and idx2 respectively.
;;; used in bk.fix-rejoin below
(defun find-vops-pair(idx1 idx2 cycle)
   `(,(find-vop-mi idx1 cycle) ,(find-vop-mi idx2 cycle)) )

;;; given a bookkeeper (dummy) jumper, finds the real (original op) jumper
;;; assuming that the true jumper is an imediate predecessor of
;;; that it is on-trace. These assumptions should be fine, since we
;;; are interested only in rt-disamb code anyway
;;; NOT  USED
(defun find-true-jumper (jumper)
  (let ( (lst (for (i in *tr*) (filter
                   (if (&& (memq jumper (mi:lstflowsu i))
                           (oper:group (car (mi:source i))))  (then i)
                   (else (if (! (mi:source  (car (mi:lstflowpr jumper))))
                             (find-true-jumper (car (mi:lstflowpr jumper))))))
                     ))) )
  ;;;  (assert (<= 1 (length  lst)))
       (car lst)))


;;; given 2 mi's checks if they are scheduled in dep order
;;; used in bk.fix-rejoin below
(defun dependency-order? (mi1 mi2)
```

```
  (let ((cycle1 (mi:first-cycle mi1))
        (cycle2 (mi:first-cycle mi2)) )

       (|| (> cycle2 cycle1)             ;;; (strict dependency order )
           (&& (>= cycle2 cycle1)        ;;; OR (non-strict order and
               (== 'vload                ;;;     non-strict dependency)
                   (oper:group (car (mi:source mi1))))))))


;;; correct rejoin points for rt-disambiguation
;;; used in bk.fix-rejoin below
(defun correct-rejoin-points-for-rt-disambiguation ()
  (let ( (mi1 nil) (mi2 nil) )
       (for ((i1 i2) in *assertion-list*) (do
            (desetq (mi1 mi2) (find-vops-pair i1 i2 rejoin-cycle))
            (if mi1 (then
                ;;; the reason why we must test both mi1 and mi2 is that mi2
                ;;; may be above rejoin-cycle, and mi1 below (if both are
                ;;; above, they don't affect us here.
                (if mi2 (then  ;;; both mi1 and mi2 are below tentative rejoin
                    (if (! (dependency-order? mi1 mi2)) (then
                        ;;; if not  in dep order
                        (:= rejoin-cycle (+ (max (mi:last-cycle mi1)
                                                 (mi:last-cycle mi2))
                                            1)))))
                (else ;;; clearly out of order (mi2 above mi1)
                    (:= rejoin-cycle (+ (mi:last-cycle mi1) 1))))))))))


;;; This procedure comes from  tr:bookkeep.lsp
(defun bk.fix-rejoin ( jumper )
(let* ( (jumpee      (car (mi:lstflowsu jumper) ) )     ;*** Better be just 1
        (rejoin-cycle (bk.find-rejoin-cycle jumpee) )
        (rejoin-mi    (*coal* rejoin-cycle) )
        )

  ;;;***begin-insert***;;;
  (correct-rejoin-points-for-rt-disambiguation)
  (:= rejoin-mi  (*coal* rejoin-cycle) ) ;;; reset it in case rejoin-cycle has
                                         ;;; changed
  ;;;***end-insert***;;;

  ;*** Note that having a blank at the end of the schedule is very
  ;*** convenient, since we can be sure of making the rejoin.  Without
  ;*** it, we might not even be safe jumping to the last element, and
  ;*** wouldn't know where to rejoin without a big special case.

  (:= (mi:lstflowsu jumper) (list rejoin-mi) )  ;*** instead of the jumpee
  (push jumper (mi:lstflowpr rejoin-mi) )

  ;*** Finally, figure out which mops were scheduled too early for the
  ;*** rejoin, but really need to be jumped to.  Make sure that they
  ;*** are copied before the rejoin, so they get executed too.  After
  ;*** the copies are made, we splice in the def and partial schedule
```

```
    ;*** of multi-cycle operations that spanned the join point that the
    ;*** code generator told us about.  Each join edge then looks like:
    ;***
    ;*** off-trace --> join-copies --> use --> partial-schedule --> rejoin-mi
    ;***

    (loop (incr inx from (mi:tracepos jumpee) to *trsize*) (do
        (if (&& (mi:first-cycle (*trarray* inx) )   ;*** scheduled?
                (< (mi:last-cycle (*trarray* inx) ) rejoin-cycle)
                ;;;***begin-insert***;;;
                ;;; get rid of if-ine's and asserts
                (remove-rt-disamb-code (*trarray* inx)) ;;;remove if returns nil
                ;;;***end-insert***;;;
                )
        (then
            (:= jumper (bk.copy-into-rejoin jumper
                                            (*trarray* inx)
                                            jumpee
                                            rejoin-mi
                                            rejoin-cycle) ) ) ) ) )
    (bk.splice-use&partial-schedule
        jumper
        (schedule:join *schedule* rejoin-cycle *all-vars*)
        rejoin-mi)
    () ) )


;;; This procedure comes from  tr:bookkeep.lsp
(defun bk.copy-into-rejoin
       ( jumper mi-to-copy old-rejoin-mi new-rejoin-mi new-rejoin-cycle )
    (let ( (copied-mi (bk.mi:copy mi-to-copy 'join) ) )

        (bk.splice (list jumper)
                   copied-mi
                   new-rejoin-mi)
        (++ *rejoin-count*)
        (if *print-copying* (then
            (bk.print-copy-before-message mi-to-copy old-rejoin-mi) ) )

        (if (cond-jumpp mi-to-copy) (then
            (loop (incr i from (mi:tracepos old-rejoin-mi)
                          to   (1- (mi:tracepos mi-to-copy) ) )
                  (initial pred-mi copied-mi)
                  (when (&& (mi:first-cycle (*trarray* i) ) ;*** scheduled?
                            (! (< (mi:last-cycle (*trarray* i) )
                                  new-rejoin-cycle) )
                            ;;;***begin-insert***;;;
                            ;;; get rid of if-ine's and asserts
                            ;;; remove if returns nil
                            (remove-rt-disamb-code (*trarray* inx))))
                            ;;;***end-insert***;;;
            (do
                (:= pred-mi (bk.splice (list pred-mi)
                                       (bk.mi:copy (*trarray* i) 'join)
```

```
                              (bk.mi:off-tracesu mi-to-copy) ) )
              (++ *rejoin-count*)
              (if *print-copying* (then
                  (bk.print-copy-before-message mi-to-copy old-rejoin-mi)))))
          ) )

      copied-mi) )



;;; *run-time-traces-number* is the number of traces in which rt disambiguation
;;; code is allowed to propagate.  This is acheived by preventing the 2
;;; bookkeep functs above to consider such code for rejoin/split compensation.
;;; When *trace-number* > *run-time-traces-number* and is-rt-code? then
;;; remove-rt-disamb-code  returns nil (which prevents the mi to be considered).
;;; The settings for  *run-time-traces-number*  can be any integer (>= 0),
;;; or nil for no restriction. The default is nil.
;;; used in bk.fix-one-flowsu and bk.fix-rejoin above
(defun remove-rt-disamb-code  (i)
   (if (&&  *run-time-traces-number*
            ;;; this condition affects the NEXT traces, not the current one.
            (> *trace-number* *run-time-traces-number*)
            (is-rt-code? i)) (then  nil) ;;; i.e. discard op
       (else t) ;;; i.e consider op
       ))

;;; returns T if i is a mi for a RT source operation (if-ine or assert-ine)
;;; (used in remove-rt-disamb-code?)
(defun is-rt-code? (i)
   (||
     (match-rt-pattern (car (mi:source i)))
     (== 'assert-ine (caar (mi:source i))) ) )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; taken from  exp:skex.lsp
(defun skex ( &optional (code          *skex.code*)
                        (actual-params *skex.actual-params*) )
    (skex.initialize)

    (:= *skex.code*          code)
    (:= *skex.actual-params* actual-params)

    (format t "~&~%")
    (options.print)
    (format t "~&~%")

    (unwind-protect
        (let ()

            (if *skex.time-functions?* (then
                (apply 'time-functions *skex.timed-functions*) ) )
```

```
            (:= *skex.seq-naddr-unoptimized*
            (compile-tiny-lisp *skex.code*) )

            (:= *skex.seq-naddr-optimized*
                (fg.analyze&optimize *skex.seq-naddr-unoptimized*) )

            ;;;***begin-insert***;;;
            (set-up-runtime-disambiguation)
            ;;;***end-insert***;;;

            (if *skex.compact?* (then
                (make-get-s *skex.seq-naddr-optimized*)
                (compact)
                (:= *skex.par-naddr* (mis->pnaddr *s*) ) )
            (else
                (:= *skex.par-naddr* *skex.seq-naddr-optimized*)
                (:= *skex.seq-naddr-optimized* *skex.seq-naddr-unoptimized*)))

            (remove-assert-ines) ;;; they are really compiler directives
            (skex.run-programs)

            (if *skex.time-functions?* (then
                (format t "~&~%")
                (print-function-times) ) )

            () )

        (if *skex.time-functions?* (then
            (untime-functions) ) ) )
    () )

(defun remove-assert-ines ()
   (:= *skex.par-naddr*
       (for (i in *skex.par-naddr*) (save
            (for (j in i) (when (|| (! (listp j))
                                    (&& (listp j)
                                        (! (== 'assert-ine (car j))))) (save
                j))))))

;;; auxiliary function  for skex above
(defun reset-runtime-disambiguation-temps ()
   (putprop '*tmp* 0 'counter) )

;;; auxiliary function  for skex above
(defun gen-runtime-disambiguation-temps ()
   (symbol:cat
       '*tmp*
       (putprop '*tmp* (+ (get '*tmp* 'counter) 1) 'counter)) ) )

;;; auxiliary function  for skex above
(defun reset-runtime-disambiguation-label ()
   (putprop '*rdl* 0 'counter) )

;;; auxiliary function  for skex above
```

```
(defun gen-runtime-disambiguation-label ()
   (symbol:cat
       '*rdl*
       (putprop '*rdl* (+ (get '*rdl* 'counter) 1) 'counter)) ) )


;;; coordinates the changes necessary for runtime disambiguation (see below)
;;; auxiliary function  for skex above
(defun set-up-runtime-disambiguation ()
   (let ( (temp (copy *des.non-zero-dexprs*)) )
     (:= *assertion-list* nil)  ;;; used in bk.fix-rejoin, dependency-order?
                                ;;; set in insert-runtime-disambiguation-code
     (setq-if-unbound *peephole-optimization-for-rt-disamb* )
     (setq-if-unbound *run-time-traces-number* nil)

     (FORMAT T "~%~%OPTIONS FOR RUNTIME DISAMBIGUATION")
     (FORMAT T "~%*PEEPHOLE-OPTIMIZATION-FOR-RT-DISAMB* = ~S"
             *peephole-optimization-for-rt-disamb* )
     (FORMAT T "~%*RUN-TIME-TRACES-NUMBER* = ~S~%~%"
             *run-time-traces-number*)

     ;;; the full name is too long, we use a shorter one in the program
     (:= *skex.pho-rtdc* *peephole-optimization-for-rt-disamb* )
     (reset-runtime-disambiguation-temps)
     (reset-runtime-disambiguation-label)
     (set-up-continuations *skex.seq-naddr-optimized*)
     (insert-runtime-disambiguation-code *skex.seq-naddr-optimized* nil nil)
     (clear-continuations)
     (:= *FG.ELIMINATE-COMMON-SUBEXPRESSIONS?-TEMP* ;;; save
         *FG.ELIMINATE-COMMON-SUBEXPRESSIONS?*)
     (:= *FG.ELIMINATE-COMMON-SUBEXPRESSIONS?* nil) ;;; don't redo it

     (:= *skex.seq-naddr-optimized*
         (fg.analyze&optimize *skex.seq-naddr-optimized*) )

     (:= *FG.ELIMINATE-COMMON-SUBEXPRESSIONS?*      ;;; restore
         *FG.ELIMINATE-COMMON-SUBEXPRESSIONS?-TEMP*)

     (:= *des.non-zero-dexprs*  ;;; only originals allowed in assertions lst
         (for (i in *des.non-zero-dexprs*) (when (member i temp))
              (save i)))))




;;; traverses the program-graph, along each possible path, and inserts
;;; runtime disambiguation code for unresolved ambiguities.
;;; For now, paths go thru loop-starts/ends, and stop at trace-fences/stop stats
;;; auxiliary function  for skex above
(defun insert-runtime-disambiguation-code (prg vops-list asserted)
  (let ( (operation (car prg))
         (tmp nil)
```

```
            (if-ptr nil) )
        (prog ()

         lbl ;;; (print operation)

          (selectq (oper:group operation)

              ( (vload vstore)
                  (:= tmp (cdr prg))

                  (for (i in vops-list)
                       (when (&& (! (member '(,(oper:part i 'index)
                                              ,(oper:part operation 'index))
                                            asserted))
                                 (! (member '(,(oper:part operation  'index)
                                              ,(oper:part i 'index))
                                            asserted))

                                 (== (car (disambiguate i operation))
                                     'maybe)))
                       (do (insert-code if-ptr
                                        (oper:part i 'index)
                                        (oper:part operation 'index))
                           (:= asserted (:= *assertion-list*
                               (append1 asserted
                                        '(,(oper:part i 'index)
                                           ,(oper:part operation 'index))))))

                  (:= if-ptr nil)
                  (:= vops-list (append1 vops-list operation))
                  (:= prg tmp) ;;; reset to real continuation
                  (:= operation (car prg))
                  (go lbl) )

              ( (stop trace-fence))

;             ( 'loop-start         ;;; loops are considered separately since
;                                   ;;; TS doesn't pick traces across loops
;(format t "entering loop:~s~%" operation)
;                 (insert-runtime-disambiguation-code
;                         (cdr prg)
;                         nil asserted )
;(format t "exiting loop:~s~%" operation)
;                 (:= prg (skip-loop prg))
;                 (:= operation (car prg))
;                 (go lbl))

              ( (if-then-else cond-jump)
                  (if (desetq (v1 v2) (match-rt-pattern (car prg))) (then
                      (if *skex.pho-rtdc* (then  ;;; setup for peephole opt of rtd
                          (:= if-ptr prg)
                          (:= prg (cddddr prg))
                          (:= operation (car prg))
                          (go lbl))))
```

```
              (else
                  (insert-runtime-disambiguation-code
                            (get (oper:part operation 'label1) 'continuation)
                            vops-list asserted )
                  (insert-runtime-disambiguation-code
                            (get (oper:part operation 'label2) 'continuation)
                            vops-list asserted ) ) ) )
          ( 'goto
              (:= prg  (get (oper:part operation 'label1) 'continuation))
              (:= operation (car prg))
              (go lbl) )

          ( 'nil ) ;;; can arise only from generated code

          ( t
              (:= prg (cdr prg))
              (:= operation (car prg))
              (go lbl) ) )
      ) ) )


;;; auxiliary function  for skex above
(defun skip-loop (p)
 (let ((ctr 0))
     (loop (while p)
           (do (selectq (oper:group (car p))
                        ('loop-start (++ ctr))
                        ('loop-end   (-- ctr))
                        (t ))
               (:= p (cdr p)))
           (until (== ctr 0))
           (result p))))


;;; insert-code does the actual insertion
;;; auxiliary function  for skex above
(defun insert-code (if-ptr i j) (format t "insert-code for op:~s~%" operation)
   (let ( (lbl1 (gen-runtime-disambiguation-label) )
          (lbl2 (gen-runtime-disambiguation-label) )
          (tmp  (car prg)) )
       (if if-ptr (then
           (peep-hole-optimize if-ptr)
           (set-prg-to-next-if-ine)) ;;; for repeated phole optim.
       (else (if (&& (desetq (v1 v2) (match-rt-pattern (car prg)))
                   *skex.pho-rtdc*) (then    ;;; if-inserted in this round
               (peep-hole-optimize prg)
               (set-prg-to-next-if-ine))
             (else
               (insert-sequence) ) ) ) ) ) )


;;; auxiliary function  for skex above
(defun set-prg-to-next-if-ine ()
```

```
  (loop (while (! (match-rt-pattern (car prg)))
        (do (:= prg (cdr prg)))))


;;; auxiliary function  for skex above
(defun peep-hole-optimize (p)     (format t "peep-holeoptimizer~%")
   (let ( (tmp1 (gen-runtime-disambiguation-temps))
          (tmp2 (gen-runtime-disambiguation-temps))
          (tmp3 (gen-runtime-disambiguation-temps))
          (lst nil) )
      (pop (cdr p)) ;;; remove label1
      (loop (while (== (caadr p) 'assert-ine)) ;;; save a
           (do (:= lst (append1 lst (pop (cdr p))))))
      (pop (cdr p)) ;;; remove label2

      (rplaca p '(ine ,tmp1 ,i ,j))        ;;; starting test in the series
      (push '(label ,lbl2) (cdr p))        ;;; second label
      (for (i in lst) (do                  ;;; restore previous assertions
           (push i (cdr p))))
      (push '(assert-ine ,i ,j) (cdr p))  ;;; new assertion

      (push '(label ,lbl1) (cdr p))
      (push '(if-ine ,tmp3 0 .9 ,lbl1 ,lbl2) (cdr p))
      (if (!=p v2 0) (then
          (push '(iand ,tmp3 ,tmp1 ,tmp2) (cdr p))
          (push '(ine ,tmp2 ,v1 ,v2) (cdr p)))
      (else
          (push '(iand ,tmp3 ,tmp1 ,v1) (cdr p))))  ) )


;;; checks if p is of the form "(if-ine v1? v2? .9 * *)" and retuns (v1 v2)
;;; auxiliary function  for skex above
(defun match-rt-pattern (p)
   (let ( (v1 nil)  (v2 nil) )
        (if  (&& (== (car p) 'if-ine)
                 (= (cadddr p) .9)
                 (:= v1 (cadr p))
                 (:= v2 (caddr p)))  (then
             '(,v1 ,v2)) )))


;;; auxiliary function  for skex above
(defun insert-sequence ()
   (rplaca prg '(if-ine ,i ,j .9 ,lbl1 ,lbl2))
   (push tmp (cdr prg))
   (push '(label ,lbl2) (cdr prg))
;;;(push '(assign jnk 1) (cdr prg))  ;;; just to make if-handling correct
                                     ;;;doesn't help, but problem is in bookkpr?
   (push '(assert-ine ,i ,j) (cdr prg))
   (push '(label ,lbl1) (cdr prg)) )
```

```
;;; for each label, set prop "continuation" to its following operations
;;; auxiliary function  for skex above
(defun set-up-continuations (s)
   (:= *skex.labels*  nil)
   (loop (initial i s)
         (while i)
         (do
            (if (== (oper:operator (car i)) 'label) (then
                (putprop (oper:part (car i) 'label1)
                         (cdr i)
                         'continuation)
                (push (oper:part (car i) 'label1)
                      *skex.labels*))))
         (next i (cdr i))) )


;;; auxiliary function  for skex above
(defun clear-continuations ()
   (for (i in *skex.labels*) (do (remprop i 'continuation))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Taken from tr:compact.lsp
(defun com.schedule ()

    ;;;***begin-insert***;;;
    (activate-run-time-assertions )
    ;;;***end-insert***;;;

    (:= *schedule*
        (generate-code
         '( (() () () ,*all-vars* ,*all-vars*)
            ,for (mi in *tr*) (save
                                 '(,(car (mi:source mi) )
                                   ,(mi:jump-dir mi)
                                   ,mi
                                   ,*all-vars*
                                   ,*all-vars*) ) ) ) ))

    (:= *schedsize*    (schedule:length *schedule*) )
    (:= *total-cycles* (+ *total-cycles* *schedsize*) )

    ;;;***begin-insert***;;;
    (deactivate-run-time-assertions) ;;remove assertions valid only for this tr.
    ;;;***end-insert***;;;
    )


;;; used in activate-run-time-assertions below
(defun supressed-rt-code? ()
    ;;; if stopping of rt-code, and it is already active for CURRENT TRACE
    (&& *run-time-traces-number*
```

```
        ;;; or not yet triggered
        (>  *trace-number*
            (+ *run-time-traces-number* 1 ) ) ) )


;;; Asserts inequalities for run-time disambiguation consistent w/ current trace
;;; used in com.schedule above
(defun activate-run-time-assertions ()
   (:= *tr.assertions-set*
       (for (mi in *tr*) (when (== (caar (mi:source mi) )
                                   'assert-ine) )
       (filter

        ;;; gross hack to avoid annoying problem of asserts moving past if's
    ;;; (if (> (length *tr*) 1) (then (:= *tr* (delq mi *tr*))))

   ;;;(if (! (supressed-rt-code?)) (then
          (car (de:assert-not-equal
                  (stat:operand-derivation
                    (hash-table:lookup *dis.oper:stat* (car (mi:source mi) ) )
                    (stat:part (hash-table:lookup *dis.oper:stat*
                                         (car (mi:source mi) ) ) 'read1) )
                  (stat:operand-derivation
                    (hash-table:lookup *dis.oper:stat* (car (mi:source mi) ) )
                    (stat:part (hash-table:lookup *dis.oper:stat*
                                         (car (mi:source mi) ) ) 'read2) ))))
       )))))


;;; After current trace is compacted, remove assertions valid only for it.
;;; used in com.schedule above
(defun deactivate-run-time-assertions ()
    (for (i in *tr.assertions-set*) (do
         (:= *des.non-zero-dexprs* (delq i *des.non-zero-dexprs*) ) ) ) )
```

References

1. A. V. Aho and J. D. Ullman. *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 -- SIGARCH 9(3), May, 1981, pp. 291-302.

3. U. Banerjee, S. C. Chen, D. J. Kuck and R. A. Towle. "Time and Parallel Processor Bounds for Fortran-Like Loops." *IEEE Trans. on Computers 28,* 9 (September 1979), 660-670.

4. Uptal Banerjee. Speedup of Ordinary Programs. Tech. Rept. UIUCDS-R-79-989, University of Illinois Department of Computer Science, Oct., 1979.

5. S. C. Chen and D. J. Kuck. "Time and Parallel Processor Bounds for Linear Recurrence Systems." *IEEE Trans. on Computers 24,* 7 (July 1975), 701-717.

6. Comte, D., Hifdi, N., and Syre, J.C. "The data driven LAU multiprocessor system: results and perspectives." *Proc. IFIP Congress 80* (Oct. 1980), 175-180.

7. 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.

8. 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, 1974, pp. 126-132.

9. John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* Ph.D. Th., YALE, July 1984. Expected

10. J. A. Fisher. The optimization of horizontal microcode within and beyond basic blocks: An application of processor scheduling with resources. U.S. Department of Energy Report COO-3077-161, COURANT, Oct., 1979.

11. J.A. Fisher. An Effective Packing Method for Use with 2^n-way Jump Instruction Hardware. 13th Annual Microprogramming Workshop, SIGMICRO, Nov., 1980, pp. 64-75.

12. J. A. Fisher. "Trace scheduling: A technique for global microcode compaction." *IEEE Transactions on Computers c-30,* 7 (July 1981), 478-490.

13. J. A. Fisher. Very long instruction word architectures and the ELI-512. Tech. Rept. 253, Yale University Department of Computer Science, Dec., 1982. Also appeared in 10th Annual International Architecture Conference, Stockholm, JUNE '83

14. C. C. Foster and E. M. Riseman. "Percolation of code to enhance parallel dispatching and execution." *IEEE Trans. on Computers 21,* 12 (Dec. 1972), 1411-1415.

15. William H. Harrison. "Compiler analysis of the value ranges for variables." *IEEE Trans. on Software Eng. SE-3,* 3 (May 1977), 243-250.

16. Hearn, A.C. Reduce-2 User's Manual. Technical Report, 1973.

17. J. Hennessy, N. Jouppi, S. Przbyski, C. Rowen, T. Gross, F. Baskett and J. Gill. MIPS: A Microprocessor Architecture. 15th annual microprogramming workshop, SIGMICRO, Oct., 1982, pp. 17-22.

18. Heuft, R.W. and Little, W.D. "Improved Time and Parallel Processor Bounds for Fortran-like Loops." *IEEE Trans. on Computers 31*, 1 (Jan. 1982), 78-81.

19. Keller, R.M., Lindstrom, G., and Patil, S. A loosely-coupled applicative multiprocessing system. AFIPS, AFIPS, June, 1979, pp. 613-622.

20. Kennedy, K. Automatic Translation of Fortran Programs to Vector Form. Tech. Rept. 476-029-4, Rice University, Oct., 1980.

21. Knuth, Donald E.. *The Art of Computer Programming.* Volume 1: *Fundamental Algorithms.* Addison-Wesley, Reading,Massachusetts, 1968.

22. Kuck,D.J.; Muraoka,Y.; and Chen, S.-C. "On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup." *IEEE Trans. Comp. C-21*, 12 (December 1972), 1293-1310.

23. Kuck,D.J. . Parallel Processing of Ordinary Programs. In *Advances in Computers Vol. 15*, Rubinoff,M. and Yovits,M.C., Eds., Academic Press, 1976, pp. 119-179.

24. Kuck, D.J, Khun,R.H., Padua,D.A., Leasure,B. And Wolfe,M. Dependence Graphs and Compiler Optimizations. Proceedings of POPL 81, ACM, Jan., 1981, pp. 207-218.

25. Kuhn, R.H. *Optimization and Interconection Complexity for: Parallel Processors, Single Stage Networks and Decision Trees.* Ph.D. Th., University of Illinois at Urbana-Champain, Feb. 1980. Report No. 80-1009

26. Kung, H.T. "Why Systolic Architectures?" *IEEE Computer 15*, 1 (Jan. 1982), 37-46.

27. L. Lamport. "The Parallel Execution of DO Loops." *Comm. ACM 17*, 2 (Feb. 1974), 83-94.

28. Z. Manna, S. Ness and J. Vuillemin. "Inductive Methods for Proving Properties of Programs." *CACM 16*, 8 (Aug. 1973), 491-502.

29. Alexandru Nicolau and Joseph A. Fisher. Using an oracle to measure parallelism in single instruction stream programs. 14th annual microprogramming workshop, SIGMICRO, Oct., 1981, pp. 171-182.

30. D. A. Padua, D. J. Kuck, and D. H. Lawrie. "High speed multiprocessors and compilation techniques." *IEEE Trans. on Computers 29*, 9 (Sept. 1980), 763-776.

31. D.A.Paterson and C.H.Sequin. "A VLSI RISC." *Computer 15*, 9 (September 1982), 8-21.

32. E. M. Riseman and C. C. Foster. "The Inhibition of Potential Parallelism by Conditional Jumps." *IEEE Trans. on Computers 21,* 12 (Dec. 1972), 1405-1411.

33. J. C. Ruttenberg. *Delayed Binding Code Generation for a VLIW Supercomputer.* Ph.D. Th., YALE, June 1984. Expected

34. G. S. Tjaden and M. J. Flynn. "Detection and parallel execution of independent instructions." *IEEE Trans. on Computers 19,* 10 (Oct. 1970), 889-895.

35. Treleaven, P.C., Brownbridge, D.R., Hopkins, R.P. "Data-driven and demand-driven computer architectures." *Computing Surveys 14,* 1 (March 1982), 93-143.

36. Wolfe, M.J. *Optimizing Supercompilers for Supercomputers.* Ph.D. Th., University of Illinois at Urbana-Champain, Oct. 1982. Report No. 82-1105