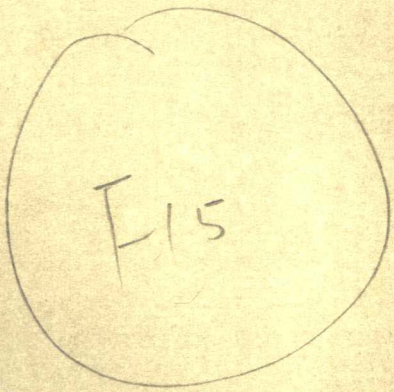
The 16th Annual Microprogramming Workshop

PROCEEDINGS



Prepaid Price
Members $18.75
Nonmembers $37.50

Library of Congress Number 83-81667
ISBN 0-89791-114-8 (paper)
ISBN 0-8186-4494-X (microfiche)
ISBN 0-8186-8494-1 (casebound)

Additional copies may be ordered from:

ACM Order Department
P.O. Box 64145, Baltimore, MD 21264
ACM Order No. 533831

IEEE Computer Society
P.O. Box 80452
Worldway Postal Center
Los Angeles, CA 90080
Computer Society Order No. 494

IEEE Service Center
445 Hoes Lane, Piscataway, NJ 08865
IEEE Catalog No. 83CH1925-7

Preface

Some years ago, one of my mentors took a hard line on the study of microprogramming, which I paraphrase as follows:

> Microprogramming is just an implementation technique, a way to get the end built, rather than an end in itself. There is no more reason to have a Workshop on Microprogramming than to have one on, say, binary decoders.

In an important sense, this is true. There is no good reason I can think of for having a workshop on any well-understood implementation technique, which leaves us with Micro-16, part of the continuing effort to understand microprogramming.

The production of any artifact typically goes through several stages: art, craft, engineered routine, and mechanization. This holds equally well for clay pots, binary decoders, and microcode. Any implementation technique that remains an art is not going to succeed without the artist; artists are generally in short supply, and difficult to create. Craftsmen are easier to train: by making microprogramming a craft we increase its applicability. If we can master the routine production of microcode, and automate that process, then we will indeed have a well-understood technique, and no more workshops; the binary decoder analogy will finally apply.

This year's workshop has an emphasis on firmware engineering and methodology, which indicates to me that while we may have mastered the art of microprogramming (well, now and then, one or two of us), and maybe even the craft of microprogramming, we have not yet mastered the routine production of microprograms. But a look at the papers indicates we're making progress.

Putting together a workshop requires a great deal more than booking a site and collecting a bunch of papers and people. As general chairman, I found myself constantly being surprised at how much the organizing committee had to do, and constantly gratified that the job was being done. Without the help of these good people, there wouldn't be a workshop. The institutional support we have received has also been indispensable, and is gratefully acknowledged. Special thanks go to Karen Jones and Marie DiLabio for deciphering my scrawl, and to Linda Torchia for creating the logo and producing the artwork well and quickly.

Bill Hopkins
General Chairman

Foreword

As with its predecessors, MICRO-16 continues to explore the art, science, and technology of microprogramming. As the reader of these Proceedings will note, a major emphasis of the present Workshop is the evolving discipline of firmware engineering which may be 'defined' as the application of scientific principles to the design, development, and production of microcode. It is pleasing to note, however, that some of the more 'classical' problems, related to architectural issues and migration, continue to draw attention.

As program chairman it is my pleasure to acknowledge the support of a number of persons who helped in putting together the technical program. I would like to thank in particular, the referees and the members of the Program Committee who, working within a very tight schedule, managed to review and produce meaningful reports on the submitted papers. These referees are listed on a separate page. It was a pleasure to work with the general chairman, Bill Hopkins. Scott Davidson and Rich Belgard provided psychological and logistic support when most needed. Cathy Pomier provided her usual cheerful and competent secretarial help. And, finally, my thanks to the authors who, of course, are the central focus of this Workshop.

Subrata Dasgupta
Program Chairman

Session I
Keynote Address

Chairman

M.J. Flynn
Stanford University

TOWARDS BETTER INSTRUCTION SETS

Michael J. Flynn*

Stanford University, Stanford, CA and Palyn Associates, San Jose, CA

Abstract

An effectively designed instruction set is the result of many considerations. These include not only obvious measures such as code size, performance and implementation cost, but also issues such as compatibility and especially design complexity.

In an effort to reduce the design complexity, flexible or universal base designs have been used to realize various instruction sets or source problem requirements.

Using either a dedicated or universal host, language oriented DEL architectures may offer some significant performance advantages over more traditional architectures.

Towards Better Instruction Sets

Probably few areas of computer system design are as controversial as the architecture or instruction set of a processor. As Professor Wilkes observed in last year's keynote address (1), the instruction set is a primary force and influence upon the microprogrammer. Many things are implied by instruction set, including software functionality, software structure, hardware cost, design effort, etc.

A great deal of recent discussion has centered on the merits of reduced vs complex instruction sets. While this focus may be useful, I believe that there are other issues in architecture at least as important as RISC vs CISC. Some of these include dedicated host designs vs universal host designs; design time vs performance; evaluation of various universal host proposals; language oriented vs host oriented instruction sets. We will look at some of these after considering the factors influencing instruction set design.

* work supported in part by Army Research Office Durham, under contract #DAAG29-82-K-0109

A Model (2)

The instruction set lies at the boundary between compilation and interpretation. One can regard a computer as consisting of three parts: an architecture or instruction set, a storage, and an interpretive mechanism. The interpretive mechanism is programmed to execute the instruction set and cause the specified state transitions in the memory. Since the interpretative mechanism may itself be a machine - with storage and (micro) instruction set we have a notion of two machines, the host or micromachine doing the interpretation of the image machine or the instruction set processor.

The question of whether the image machine is "high level" or "low level" is deceptively simple sounding. Presumably a low level architecture requires few host instructions to execute its state transitions, while a high level machine requires many more. This, of course, ignores the potential sophistication of the host itself, what resources it has and what constraints have been placed upon it. In the absence of being able to standardize on a particular host configuration, I prefer to classify instruction sets as being either language oriented or host oriented with the distinction being that a language oriented architecture describes storage transformations in semantics similar to those used in a high level language source program while a host oriented architecture uses objects present or presumed to be present in the host; memory cells, registers, etc. One might expect that a language oriented instruction set would necessarily be more complex than a host oriented one, i.e. would take more host cycles to interpret a given instruction. This is not necessarily true, as many recent host oriented instruction sets have extensive interpretation requirements (therein lies the basis of much of the RISC vs CISC argument (3)).

What Makes a Good Instruction Set?
Primary Factors

There have been many proposals to evaluate instruction sets (4), usually on a quantitative basis. Unfortunately, these evaluations may be misused since the evaluation process usually restricts the scope of test material to small problem state programs. Whatever the limitations

of such measurements they do form at least a partial basis for instruction set evaluation. Some of these measures include:

1. Static measures - In the absence of other considerations, smaller static code size is better. More concise code can be emitted more rapidly by a compiler, should have better locality in the memory hierarchy and require less memory bandwidth for the instruction stream.

2. Compilation time - An item frequently overlooked in many evaluations is the time it takes to compile a source program into an instruction set. A typical processor spends perhaps half its time in compilation and the other half in problem execution. While compile time implications are just as important as run time implications, it is difficult to measure the influence of instruction sets on a compilation process. One compiler may attempt elaborate optimization while another one is "compile and go." Some generalities are possible: architectures that require register usage optimization and large program size, for example, would require more compilation time than those without the same requirements. However, these generalities are of limited use. The evaluation of an instruction set for ease of compilation is confused by the complexity of the compilation process itself. Compilers differ in their debugging functionality, their attempt at source to source optimization (rearranging the original source program in its best possible form), and simply the skill of the compiler writer.

3. Run time - Dynamic measures of an architecture include such artifacts as number of instructions executed, number of data references required for reads and writes, and memory traffic required as measured in bytes transferred. Ultimately, one would like to include a measure of the number of host cycles required to execute an image program. Like compile time this is difficult to do on a comparative basis, since different hosts have different degrees of support for various image machines.

4. Predictability - Related, at least indirectly, to both compilation time and run time is the predictability of finding (decoding) and executing fragments of an instruction. Predictability comes at the expense of code size since information is by definition directly related to uncertainty. However, many architectures are created in such a way as to require needless serial interpretation of fields as part of the execution process. The interpretation of field B may depend upon the result of the interpretation of field A. The use of variable width codes to encode a given field is another difficulty. Huffman codes, for example, require a serial inspection of the contents of a field on a bit by bit basis to determine even the width of the field. Predictability in an instruction set allows successful overlapping and lookahead on the part of the host in the interpretation of the image instruction stream.

In addition there are other types of dependencies which inhibit execution predictions by the host.

1. Interinstruction dependencies where the execution of the present instruction depends upon the past instruction. While much of this is unavoidable, several of the worst interlock difficulties in high speed machines such as store into the instruction stream can be eliminated by disciplined instruction set design.

2. Process to process dependencies - here again a well defined call and return mechanism minimizes the process entry costs.

5. Transparency - by definition transparency is a property of the architecture that allows it to represent source state transitions on a one for one basis. If program execution is viewed as a sequence of non-interruptible atomic actions, then at end of each non-interruptible event the transparent interpreting machine has the same state as that specified by the program being interpreted. Clearly an optimizing compiler destroys transparency between the source and the image machine. A high speed host that executes instructions out of order destroys transparency between the image instruction and the host. Transparency, as such, has not been much sought after in either architecture or compiler development efforts. Its advantages are subtle. It eases the burden - perhaps makes possible - program verification. It is also a valuable property in verifying the coordination of the execution of concurrent processors. Several paradoxes in concurrent execution arose from a failure in transparency.

While the value of transparent execution may be debatable, it is clear that language oriented architecture can more easily support such execution than host oriented instruction sets (regardless of their level of complexity).

Secondary Factors Influencing the Design of Instruction Sets

In addition to the straight forward tradeoffs between compiler technology and host technology, there are a number of additional, seemingly secondary issues whose importance frequently overwhelms the primary and more academic considerations mentioned in the previous section. These secondary issues include such

issues as compatibility, environmental stability, design time and technology.

Compatibility

All platitudes to the contrary, the primary level of transportability and compatibility of programs is the instruction set, not the higher level language. Frequently for older programs the source simply does not exist any longer. Even where the source exists the multitude of dialects of the same source language creates obvious problems.

Since most traditional instruction sets are host oriented, they were created under then current premises of host technology - the cost of a register, the speed of memory, etc. The more successful an architecture is, the more implementations it has, the longer its life, the more certain the need for continued compatibility, and finally the more difficulty in providing competitive measures of static and dynamic performance; an obvious problem in comparing architectures.

The Environment

Processes can be designed for single applications or for a universe of applications. In an age of "cheap" hardware it seems obvious that hardware should be dedicated toward an environment insofar as the application is stable and known. To use a universal processor will naturally give a poorer performance. On the other hand, economy of manufacturing scale allows such processors to be produced at extremely low costs. The cost for the design can be amortized over a much larger quantity of processors. The hardware may be cheap, but the design costs are not, unless the applicability is vast. Of course, this leads to an interest in universal structures which we will discuss in a later section.

Ever recognizing that processor costs include both design and production costs (i.e. total processor costs), one must also consider the total system cost. Memory costs, I/O, power supply, and housing each typically exceed the cost of a microprocessor and thus dominate the total cost picture.

Design Time

Complicated architectures require complicated development programs for both design and development, verification and production. The recent interest in academic institutions in reduced instruction sets stems as much from the possibility of being able to do a design of a simple architecture and being unable with the same meager resources to accomplish the implementation of a more complex instruction set using the same host technology. Here again universal structures may used to ease the design problem at the expense of the primary performance measures.

Technology

Technology provides both a direct and an indirect influence on the architecture and its resultant performance. Presently chip area constraints require an implementation to fit a fixed area and performance may be severely limited in order to force an a priori defined instruction set to fit on a particular chip. With conventional packaging techniques, pin constraints may severely restrict access to memory, placing a premium on those architectures which make best use of memory bandwidth. Just as System 370 was a child of its times in its register and memory specification, present microprocessor technology constraints are similarly products of today's limitations. Upcoming wafer technologies which package a complete wafer rather than a chip may radically alter much of the current thinking about chip oriented or area constrained processor architecture.

Three Universal Hosts

There are various ways a computer designer can create host structures without committing the design to a particular image architecture and environment. The basic difference among approaches arises from tradeoffs among compiler technology, design time, and compile and run time performance. Three more or less universal (or at least general purpose) host structures are:

1. The universal host machine (UHM) is an interpretive machine designed to emulate various high level architectures. The architecture can be specialized for particular environments, such as a particular high level language. The UHM's limitation is that it takes additional cycles to interpret the unsupported image architecture.

2. The universal executing host (UEM) relies on compiler technology to adapt the universe of environments to the processor, the IBM 801 (5) or the RISC (3) are examples of the UEM approach. From our point of view, the high level language source is compiled into a type of microcode.

3. Gate array. Initially a gate array may seem to be a strange type of host, yet when it is personalized with wiring layers - a relatively simple design process - it emulates a desired image machine. Like all universal approaches the gate array pays a price for its flexibility. A given area of silicon

must be depopulated by at least a factor of two - probably more than 4 - when compared to a custom lay out.

Each of these approaches pays a price for their flexibility, but the price is paid in different ways and at different times. In each of the above cases a single host design is adapted to multiple environments by either compiler, interpreter, or routing technology.

UEMs and UHMS of recent vintage are much more similar than different. They usually share at least the following characteristics:

1. A large read/write microstore, accessible in one internal cycle. This microstore may be used in different ways. In the UHM it would be used for microprogram storage and perhaps cache, whereas in the UEM it is used almost exclusively for cache.

2. Main memory is a block oriented device for bulk storage in a multi-level storage hierarchy. Movement of environments from remote executable levels of storage may require support ranging from simple to elaborate.

Indeed the instruction sets at the host level are quite similar. They are both designed to be executed in one ALU cycle controlling a small number of working registers. A recently designed VLSI UEM at Stanford called MIPS (6) uses an instruction format with some limited horizontal parallel capabilities. Indeed it has an instruction format very similar to the Stanford EMMY (7), a 32 bit UHM designed for experimental purposes and now in use at Stanford for several years. The MIPS processor is particularly interesting as a state of the art UEM. It has an overlapped organization with a four stage pipeline; the interlocks are handled by the compiler. The EMMY design has a three stage pipeline with interlocks handled by the emulator (interpreter).

The most significant difference between the UEM and the UHM environment is the way the microstorage is used: in the UHM there is a separate micro-program storage. UHMs and UEMs are both improved by use of a cache type high speed storage for data and programs. The UHM simply pays the price of larger program static size with a larger cache, whereas the UEM pays the price of the micro program storage.

The limited differences between UEM and UHM might well argue for a Universal machine (UM) with a reconfigurable microstorage to be (partially) used as either micro-program storage or cache.

In a recent experiment at IBM, the System 370 architecture was implemented in a gate array (8) technology. The processor chip consisted of an all bipolar gate array using 4,923 bipolar gates out of a possible (available) 7,640. The chip was a 49mm^2 die with 200 I/O pins. A gate array provides an interesting alternative to personalizing an architecture in contrast to the preceding two approaches which customize through more traditional software. Actually the complete System 370 consisted of multiple chips consisting of the gate array processor chip, a control store ROM chip, a register chip, and main memory chips. The processor implementation was area limited and thus the host was realized with only an 8 bit ALU execution (with full 24 bit address arithmetic, however). It is important to note that the purpose of this experiment was to test CAD tools and not to optimize chip design tradeoffs. The experiment was successful in that the design was accomplished in a relatively short period of time with limited manpower. In an experiment organized to develop a true custom architecture, perhaps different chip arrangements would emerge perhaps using custom macro ALUs and on chip micro-storage of some smaller size.

Custom (Language Oriented) Architectures

At Stanford in the past several years, we have developed what we call the DEL - Directly Executed Language - approach to high level architecture. DEL is actually a misnomer since the source HLL program is not directly executed. Perhaps a better acronym would be DCA - Direct Correspondent Architecture. Objects in the source are represented as single objects in the image program. A basic objective of this research was to find better ways to customize architectures to environments. As such, most of our studies have been limited to particular source languages. We have completed studies on Fortran, Pascal, and Cobol. The Fortran and Pascal architectures (called Deltran (9) and Adept (10), respectively) were fully emulated using the Emmy UHM in our emulation laboratory. Deltran was not supported with the complete compilation facility, however, Adept is fully supported. Among the number of techniques used in the creation of DEL architectures two are particularly important:

1. A robust (or complete) set of formats
2. A contour based implementation of object specification

A robust set of formats gives concise code by eliminating both memory overhead instructions (load, store, push, pop, etc.) and redundant identifiers (as in a fixed three address format). All redundancy and overhead instructions can be eliminated with 21 formats, however, for practical purpose much smaller sets (about 8) will accomplish most of the advantages of completeness. The complete format set uses an internal evaluation stack, for example consider the statement:

A = B + C * D

In a stack machine this would be executed with the following instructions:

```
push    C
push    D
*
push    B
+
pop     A
```

This would be replaced simply by two DEL instructions.

```
F1,C,D  *
F2,A,B, +
```

Where F1 is a format that identifies the two operand sources as being explicit, i.e. contained in the instruction and the result as being stack. When the result is the top of the stack, the stack is automatically pushed. The F2 format identifies the stack as the right hand source, the left hand source and the result are explicit. When the stack is a source, it is automatically popped.

The contour model is based on a description of programming languages originally proposed by Johnson. Each procedure has its own contour and at least for dynamic languages a contour which contains data values used in the called procedure is loaded from main store into high speed contour storage at procedure entry time. For both Deltran and Adept, the specified width for an object varied from contour to contour. Its size was determined simply by using the smallest integer that contains $\log_2$ of the number of unique objects present in that particular environment or scope of definition. A procedure with 7 unique variables would have 3 bit fields to identify a variable. Addressing a particular variable consists of adding the three bit field to an environmental pointer which contains the base for the contour being executed. If the value is known, as in the case of constants and locals, it is simply contained in the contour. The first access to other variables, such as array elements will create an indirect reference, the address is determined by using a descriptor contained in the contour (9, 10).

Some Observations and Data

1. The RISC vs CISC tempest. Many traditional and fairly complex instruction sets (e.g. S/370 or VAX) include "reduced" instruction sets in their repertoire. The complexity comes about from extended functionality - managing large and segmented memory spaces, operating system support, file and character handling primitives, etc., and of course compatibility. These are not measured by small single test programs.

One may observe, however, that small, single user operating systems environments may be an increasingly important processor application (e.g. workstations).

2. Universal hosts provide a general purpose framework for matching environments to hardware. All such hosts pay a price for this flexibility - and this flexibility may exist only at design time (gate arrays - or UHMs with ROMs) or come at the expense of compile time (UEM) or require run-time overhead in loading a R/W UHM microstore.

3. Gate arrays vs UHM: System 370 emulation. A UHM of comparable bipolar technology to the 370 gate array was designed and simulated at Stanford. The UHM was a custom MSI design. Comparing this to the previously described gate array experiment:

	System 370 Chip	Comparable UHM Emulating 370
Number of Gates	4,923	11,000 (est.)
Cycle time	100 ns	125 ns (est.)
Data Paths	8^b (partially mapped)	32^b UHM
Control Store	54^b X 4^k (approx.)	32^b X 4^k
	ROM	RAM
Emulator Size (System 370)	NA	2,200 words
Av. Cycle/image Instr.	50	22

In both cases the number of gates does not include either microprogram storage or the bulk of the image registers. The estimate of 125nsec was based on T^2L Schottky MSI for the UHM. Note that if the designers of the 370 chip had 11,000 at their disposal, they would have certainly been able to realize a well-mapped host - with about 8-10 cycles per image instruction. The 11,000 gates of the UHM would fit on the same $49mm^2$ chip as the 370 in a completely custom design. The significant trade-off is not area but design effort and CAD support.

A custom designed UHM chip is another way of providing a flexible host and may in complex systems, provide superior results to gate arrays.

4. Language oriented vs Host oriented architectures. Extensive tests on EMMY emulating Adept (our Pascal architecture) illustrate the significant advantage that DELs offer over conventional host oriented architectures in both static and dynamic measures. For a suite of programs including FFT, Kalman filter and maze runs, the following data (10) is referenced to Adept (Adept = 1).

Measure	IBM/360	HP1000	P-Code
Code size	2.40	2.22	3.60
Instructions executed	3.42	3.55	3.57
Inst. bytes fetched	3.29	2.57	4.12
Data bytes fetched	5.82	4.80	5.81
Data bytes stored	20.15	12.56	9.19

If experience on EMMY is used as a guide, the DEL architectures are no more complex than familiar host oriented architectures (e.g. S370 emulator size is 2200 words while Adept emulator size is 1500 words - with comparable functionality).

5. While specialized custom designs can yield significant performance/functional advantages over a universal host based design, presently the design cost of such systems is excessive. In many ways the future of architecture is bound to the future of design tools.

6. There is a significant need for continued research in architecture - so as to know how to shape an instruction set to an environment. The "art" should be removed from computer architecture.

Indeed there are many new approaches possible, such possibilities as run-time expansion of a concisely encoded instruction into an expanded form can be done as a cache is loaded. Heuristic architectures could record an execution history of program execution to improve host performance.

Conclusions:

Many factors influence instruction set design over and above the obvious compile-execution tradeoff.

Design time is an especially serious problem as much of the potential of the technology must be "derated" to accommodate limitations of either the design tools or universal hosts.

Language oriented architectures provide an important basis for the understanding of customizing an architecture to an environment. DEL derived data seems highly promising when compared with familiar host oriented instruction sets.

References

(1) Wilkes, M. V., The Processor Instruction Set, Proc. of the 15th Workshop on Microprogramming, Pub. by - IEEE Computer Society 1982 pp 3-5.

(2) Flynn, M. J., Customized Microprocessors, Microcomputer System Design Lecture notes in Computer Science No. 126, Springer-Verlay 1981, pp 182-220.

(3) Patterson, D. A. and Ditzel, D. R., "The Case for the Reduced Instruction Set Computer," Computer Architecture News, 5, 9,3, (1980) p 25.

(4) Fuller, S. H. and Burr, W. E., Measurement and Evaluation of Alternative Computer Architectures, Computer, Oct. 77, p 24-35.

(5) Radin, G., "The 801 Minicomputer," Computer Architecture News, 10, 2, (1982)

(6) Hennessy, J. L., et al, "MIPS: A VLSI Processor Architecture," Proc. CMU Conference on VLSI Systems, Oct. 81.

(7) Davis, C., et al, Gate Array Embodies System/370 Processor, Electronics, Oct. 9, 1980, pp 140-143.

(8) Flynn, M. J., Neuhauser, C. J., and McClure, R. M., EMMY - an Emulation System for User Microprogramming, AFIPS, Vol. 44 (NCC 1975) pp 85-89.

(9) Flynn, M. J., Hoevel L., W., Execution Architecture - the Deltran Experiment, IEEE Transactions on Computers, Feb. 1983.

(10) Wakefield, S. W., Studies in Execution Architecture, Ph. D. thesis, EE Dept. Stanford University 1983.

Session II
Microcode Compaction and Optimization

SRDAG Compaction - A Generalization of Trace Scheduling to Increase the Use of Global Context Information

Joseph L. Linn (1)

University of Southwestern Louisiana
Computer Science Department
P.O. Box 44330
Lafayette, Louisiana 70504

ABSTRACT - Microcode compaction is the process of converting essentially vertical microcode into horizontal microcode for a given architecture. The conventional plan calls for a microcode compiler to generate vertical code for a given architecture and then use a compaction system to produce horizontal code, thereby greatly reducing the complexity of horizontal code generation.

This paper attempts to extend the existing techniques used to perform the compaction process. Specifically, the procedure presented generalizes the "trace scheduling" method of [Fisher81] by using more global context information in compaction decisions. A number of definitions from classical compaction are generalized to encompass this expanded scope.

Further, the paper presents two example classes of problems for which the new method outperforms the trace scheduling technique in terms of the execution time efficiency of the generated code. A number of unresolved questions are noted involving the class of global compaction procedures.

1.0 Introduction

This paper presents an extension to the trace scheduling microcode compaction method [Fisher81] called "SRDAG compaction". The history of classical microcode compaction has so far been marked by two epochs. In the first epoch, the primary thrust of research was aimed at discovering reasonable solutions to the local compaction problem. The goal of any compaction algorithm is to transform an input vertical microprogram into an equivalent compacted horizontal microprogram. For the local compaction problem, the input program is restricted to be a single (basic) block, that is a program with a single entry, a single exit, and no branches.

(1) This work is supported, in part, by a grant from the RCA Corporation.

When it was discovered that this problem is NP-complete (for the most general case where the register assignment is not known a priori) [DeWitt76], attention turned to the discovery of algorithms not guaranteed to produce the optimal compaction but whose execution time is within practical limits. A number of such algorithms have been discovered and tested [Landskov80, Davidson81]. It has been determined that these algorithms achieve solutions within a few percent of the optimal in virtually all cases. However, error bounds on the performance of these algorithms have never (to the author's knowledge) been established. In the case of one algorithm, microinstruction list scheduling [Fisher79], one would assume that the well known list scheduling bounds would carry over in a reasonable way. Thus, the first epoch in classical compaction ended with the discovery of a number of algorithms running in quadradic time achieving very good solutions to the local compaction problem.

The current epoch began with attention turning to the problem of exploiting parallelism beyond block boundaries. It was immediately apparent that the the global compaction problem could not be effectively solved by merely compacting individual basic blocks autonomously; the interplay among the basic blocks of a program exerts great influence on local compaction decisions. The solution offered by [Fisher81] is trace scheduling, a technique by which a path, or trace, of the program is essentially treated as a single block to be compacted. In this way, the scheduler has a more global view of the program and interactions among blocks on the trace are taken into account. Unfortunately, interactions with blocks off the trace are not considered. In addition, compacting the entire trace is not as general as compacting only the first block of the trace. This is because a major emphasis in trace compaction is that the trace under consideration is in some sense the most likely uncompacted path through the graph. However, moving instructions from later blocks of the trace into the first may have the effect of changing which trace through the graph is the most likely. Thus, it seems reasonable that the trace should be reselected after the first block has been compacted.

© 1983 ACM 0-89791-114-8/83/0010/0011$00.75

SRDAG compaction works in a manner that is very similar to trace compaction (1) except that the subgraph considered is a singly rooted directed acyclic graph (SRDAG) instead of a path. Moreover, the entire SRDAG is not compacted simultaneously; in fact, only the root block of the SRDAG is compacted in each iteration. The primary advantage of SRDAG compaction over trace compaction is that some important improvements are considered that are not considered when the global view is restricted to a path. For example, a particular microinstruction may be free at the top (i.e. movable to a higher block, see DEFINITION 7 below) of several blocks in the SRDAG. The SRDAG compaction algorithm prefers to move up such duplicated microinstructions whereas the trace compactor is not aware of this possible improvement.

The presentation here will assume that the program graph is a directed acyclic graph (DAG), i.e. that there are no loops. The technique used for introducing loops in [Fisher81] should apply equally well to SRDAG compaction.

2.0 The Model

This paper utilizes nearly the identical model as [Fisher81]. There are a few differences that are pointed out as the appropriate definitions are given. The most fundamental unit of execution for a computing engine is a microoperation. On a horizontal machine, several microoperations may be packed together to form a microinstruction. A basic block, or just block, is a sequence of microinstructions where only the last one is permitted to be any type of branch. Thus, we are led to the following definitions:

DEFINITION 1: For any given microengine, MOP is the set of all legal microoperations for that engine.

DEFINITION 2: For any given microengine, there is a function

is_instruction : powerset(MOP) -> Boolean.

The is_instruction function replaces the resource_compatible function of [Fisher81] in the loop-free case. Further, if I1 and I2 are microinstructions with I3 equal the union of I1 and I2, I1 and I2 can be combined exactly when is_instruction(I3). As explained in [Fisher81], this function is easily calculated using a resource vector.

---

(1) The terminology "trace scheduling" from [Fisher81] stems from the fact that list scheduling is the selected local compaction technique and from the fact that the scheduling paradigm and terminology are utilized in the presentation. This presentation does not utilize the same list scheduling basis; thus, the more generic "trace compaction" is used.

DEFINITION 3: A block is a sequence of microinstructions where only the last microinstruction is permitted to contain a branch microoperation, and no branch is permitted to any microinstruction in the sequence except the first.

DEFINITION 4: Microinstructions are assumed to operate by reading and writing certain registers (memory elements) of the microengine. If REGISTERS is the set of memory elements directly accessible by microoperations we can define:

readreg, writereg:
powerset(MOP) -> powerset(REGISTERS)

The functions readreg and writereg are properties of the particular microengine being considered. Not all of the registers need actually be memory elements; indeed registers are also introduced to model the effects of microoperations on busses and other data paths of the engine. The purpose of the readreg and writereg functions is to specify in an unambiguous way how microinstructions can be rearranged while preserving the semantic meaning of the program. A relation may be defined in terms of these functions that allows us to say that a particular microinstruction must be executed before another one. This relation is defined as follows.

DEFINITION 5: Given is a DAG D of microinstructions. For any unique microinstructions Ii and Ij so that there is a path in D from Ii to Ij:

* if the intersection of writereg(Ii) and readreg(Ij) contains some register r and there is no microinstruction Ik on any path from Ii to Ij so that r is in writereg(Ik), then Ii directly data precedes Ij.

* if the intersection of readreg(Ii) and writereg(Ij) contains some register r and there is no microinstruction Ik on any path from Ii to Ij so that r is in writereg(Ik), then Ii directly data precedes Ij.

If Ii is related to Ij by the "directly data precedes" relation, the notation is Ii << Ij. Further, the name of the transitive closure of this relation is "data precedes", i.e. if Ii <<+ Ij then it is said that Ii data precedes Ij. It is possible for there to be Ii, Ij, and Ik so that Ii << Ij, Ij << Ik, and Ii << Ik. These "transitive edges" may be removed since this does not affect the behavior of the compaction procedures.

DEFINITION 6: Given a set of microinstructions, MI, and a partial order, R, defined on MI, the set of successors of a microinstruction I in MI is given by:

successors(I) = {I' in MI | I R I'}.

Of course, if I' is in successors(I) then I' is termed a successor of I.

DEFINITION 7: Given a partial order over the set of microinstructions, microinstruction I is free at the top of a block B if it contained in B and if it is not the successor of any any microinstruction in B. Microinstruction I is free at the bottom of B if I has no successors in B.

DEFINITION 8: A register r is live locally in block B if some microinstruction in B reads register r before any microinstruction writes register r. A register r is live at the top of block B if it is live locally or if there exists a block B' so that (a) there is a path from B to B' in G, (b) no microinstruction in the path from B to B' writes register r except potentially B', and (c) r is locally live in B'.

DEFINITION 9: A program P is a five-tuple <G,edgep,startp,live_reg,cmpct> 5-tuple where

G = <Blocks,Arcs> is a directed graph with the vertices (Blocks) of the graph being blocks and the edges (Arcs) representing a possibility that control can pass between the tail of an arc and its head.

edgep : Arcs -> [0,1] is a function giving the conditional probability that control will flow to the head of the arc, given that control passes to the tail of the arc.

startp : Blocks -> [0,1] is a function giving the probability that the program will start in a given block.

live_reg : Blocks -> (subsets of registers) is a function denoting which registers are live at the beginning of any block. Live_reg need only be given for leaves initially since the compaction procedure recomputes live_reg anyway.

cmpct : Blocks -> {true,false} is a function denoting whether a given block has already been compacted.

3.0 A Global Compaction Procedure

In this section, the algorithm for global compaction is presented. The various steps of the algorithm are represented by pidgen code for expository purposes; the code given should not be understood to be a tested implementation.

Global compaction may be viewed as a procedure that transforms one program into another one. The goal of global compaction is to perform this mapping in such a way as to preserve semantics of the program and also to reduce the execution time of the program by reducing both the length of individual blocks and the number of blocks through which control passes during execution. The procedure normally begins with a program in which few, if any, blocks have been compacted. There are no reported instances where a high level language microcode compiler actually operates in this fashion. Nevertheless, if a procedural binding concept [Davidson80] is incorporated in the language, the code emitted might be precompacted. Also, in the S* family of microprogramming languages [Dasgupta78, Klassen81] supports the concept of precompacted "regions". Thus, allowing some precompacted blocks simply anticipates already emerging microcode compiler technology.

In addition, the compaction procedure also requires that the program graph G be acyclic and that the program may not begin execution in any block that has a predecessor in G. The process is an iterative one; on each iteration, at least one uncompacted block is compacted yielding a new program. Unfortunately, the compaction of one block may cause other blocks to be built in order to preserve the semantics of the program. Each iteration of the procedure may be described as follows:

```
procedure SRDAG_compaction(inout P:program);
  begin
  reduce_graph_and_compute_livereg(P);
  while some block of P is not compacted do
    begin
    select_compaction_SRDAG(D,P);
    compact_root_of_D_updating_P(D,P);
    reduce_graph_and_compute_livereg(P);
    end;
  end;
```

Several items are noteworthy. First, the steps of SRDAG compaction are essentially the same as those of the trace compaction procedure of [Fisher81]. This is hardly surprising since SRDAG compaction may be viewed as a generalization of trace compaction. One notable difference is that the bookkeeping procedure of [Fisher81] is explicitly integrated into the root compaction procedure. Second, no proof of termination is presented.

In addition, this presentation makes explicit a graph reduction procedure. In the version used here, the procedure reduce_graph_and_compute_livereg effects several important transformations. First, empty blocks are deleted and strings of blocks coalesced when possible. Second, updated live register information is computed; in the same step, useless microinstructions can be eliminated. Although the procedure as shown recomputes live register information for the entire graph on each iteration, execution time might be saved by recomputing this information only for those blocks changed or created in the current iteration and their predecessors. The implementation of this procedure is a standard topic in code generation [Aho77] and is not discussed here.

This presentation concentrates on the root compaction step. However, a few remarks are in order regarding the selection of the SRDAG to be