Dense Instruction Set Computer Architecture

submitted by

Olaf S. Schoepke

for the degree of Ph.D

of the

University of Bath

1992

Signature of Author ....[illegible]..................

Olaf S. Schoepke

Summary

Processor speed is still increasing. Memory cycle time cannot keep up with this development and is proving to be the limiting factor in high performance architectures. The gap between processor and memory speed must be bridged if future processors are to reach their full potential. In this dissertation an attempt has been made to solve the problem by increasing instruction density. A new computer architecture based on dense computer instruction sets is presented. Experimental results show that most of the compiled code can be generated using a small number of opcodes. Entropy measurements reveal very low entropy in instruction streams especially when treated as higher order Markov sources. Arithmetic coding, a coding technique capable of encoding close to the entropy bound, is used to encode and decode the instruction stream. A high level VLSI design is suggested for the architecture. Simulation results show a significant reduction in memory-processor bandwidth using a context dependent model for instruction set encoding.

Acknowledgment

Thanks are expressed to my supervisor, Jeremy Bennett, for support and advice during this research work.

My appreciation also extends to other members of the *University of Bath*: Geoff Smith for proof-reading and many helpful discussions, Robert Oddy and Francesco Arci for proof-reading, and Chris Burdorf for valuable help with my publications.

I am also grateful to my parents for their constant support during this period.

A special thank you to Elisabeth without whose moral support and encouragement I would probably never have experienced university life.

Financial support for this research work from Siemens is kindly acknowledged.

Preface

Each new generation of processing technology produces faster, dynamic RAMs with larger capacity than before. Nevertheless, the main memory cycle time has been decreasing less rapidly than the clock period for processors [Hennessy & Patterson 1990]. Although today's computer designer can choose from a wide variety of communication organizations, interaction between memory and processor remains a focal point in the design of high performance systems. As processors get faster more and more performance is lost to the memory system. A new memory-processor organization has to be evolved to supply instructions to the execution unit fast enough, and thus reduce communication delays.

In this dissertation I present a computer architecture based on dense computer instruction sets [Schoepke 1992d, Schoepke 1992g] to overcome the problem of increasing memory latency by the drastic reduction of memory-processor bus traffic through compact encoding. Dense computer instruction sets are the most promising way to reduce memory-processor bus traffic considerably, and thus reduce memory latency, to improve overall system performance. In developing this methodology an advanced architecture has been designed and its implementation simulated.

Contents

1 Introduction 1

1.1 Architectures 1

1.2 Program Size 3

1.3 Memory Latency 4

1.4 Memory–Processor Connection 4

1.5 Networking 5

1.6 Information Theory 6

1.7 Summary 6

2 Background Theory 7

2.1 Analysis of High Level Language Usage 7

2.2 Analysis of Instruction Sets 10

2.3 Computer Architectures 12

2.3.1 RISC Architecture 13

2.3.2 Shared Memory Architecture 14

2.3.3 Improving Main Memory Performance 17

2.3.4 Cache Influence 18

2.4 Instruction Execution 19

2.5 Architectural Solutions 21

2.5.1 Superscalar Designs 21

2.5.2 Primitive Based Architectures 22

2.5.3 Very Long Instruction Words 23

2.5.4 Micro-Architectural Parallelism 23

2.5.5 Distributed Instruction Set Computer Architecture 24

2.6 Theoretical Analysis of Instruction Sets 25

2.6.1 Overview 25

2.6.2 Replacement of Repeated Strings 28

2.6.3 Memory Referencing Behaviour 29

2.7 Summary 30

3 A Target Architecture for Experimental Compression 31

3.1 SPARC Architecture 31

3.2 Instruction Description 33

3.3 Performance 37

3.4 Summary 38

4 Compression Techniques 39

4.1 Non Suitable Compression Techniques 40

4.2 Potentially Suitable Techniques 40

4.2.1 Huffman Coding 40

4.2.2 A Universal Algorithm for Sequential Data Compression 42

4.2.3 Prediction by Partial Matching 42

4.2.4 Arithmetic Coding 43

5 The Implementation of Arithmetic Coding 45

5.1 The Idea of Arithmetic Coding 45

5.2 Models for Arithmetic Coding 46

5.3 Simple Character Set Example 47

5.4 The Conventional Algorithm 49

5.5 A Fast Decoding Algorithm for Arithmetic Coding 50

5.5.1 The Mathematics Behind Look-Up Tables 51

5.5.2 Implementation 53

5.5.3 Experimental Results 54

5.5.4 Summary 55
5.6 Hardware Implementation 56
5.7 Summary 56

6 Experimental Results **57**
6.1 Experiments Performed 57
6.2 Empirical Analysis 59
6.2.1 Opcodes 59
6.2.2 Opcode pairs 60
6.2.3 Registers 60
6.2.4 Constants 63
6.2.5 Branch Instructions 64
6.3 Entropy 66
6.4 Coding Results 70
6.5 Summary 72

7 Proposed Model for a Compressed Instruction Set Architecture based on SPARC **74**
7.1 Standard Execution Model 74
7.2 Decode-Execution Model for Compressed Code 76
7.3 Handling Control Transfers 77
7.3.1 Overview of the Problem 77
7.3.2 Compiler Information 80
7.3.3 Solutions 80
7.3.4 Method Applied 82
7.4 Algorithms for Encoding and Decoding 82
7.5 Cache Influence 83
7.6 Hardware Design 85
7.7 Summary 90

8 Dense Instruction Set Computer Architecture Simulation **91**

8.1 Encoding and Decoding Process 91

8.1.1 Encoding 91

8.1.2 Decoding 94

8.2 A Simulator for Dense Instruction Set Architectures 95

8.3 Simulation Results 96

8.3.1 Compression Achieved 96

8.3.2 Compression Results compared to Theory 97

8.3.3 Comparison between Zero and First Order Model 98

8.3.4 Effectiveness 99

8.3.5 The Effect of Fetch Width 100

8.4 Architecture Limits 101

8.5 Summary 102

9 Conclusion **103**

List of Figures

1-1 Memory hierarchy 3

2-1 Multiprocessor Bus Interconnection 15

2-2 Multiprocessor Crossbar Interconnection 16

2-3 Multistage Interconnection Network 16

2-4 Interleaved memory organization 17

2-5 Instruction pipeline 20

2-6 Instruction super pipeline 21

2-7 Superscalar structure 22

3-1 Instruction formats 34

3-2 SPARC register windows 36

6-1 Positive constant distribution 63

6-2 Negative constant distribution 63

6-3 Positive branch distribution 64

6-4 Negative branch distribution 65

6-5 Entropy for symbols and opcodes 67

6-6 Register entropy for the dynamic case 68

6-7 Register entropy for the static case 68

6-8 Entropy for conditional and immediate field for the dynamic case 69

6-9 Entropy for conditional and immediate field for the static case 69

7-1 Instruction execution 75

7-2 Compressed object code execution 76
7-3 C code for *if* and *for* statement 78
7-4 Assembler language for *if* and *for* statement 78
7-5 Encoding algorithm 83
7-6 Decoding algorithm 84
7-7 VLSI Design for the dense instruction set computer architecture 86
7-8 Hardware design for arithmetic coding 87
7-9 Address decoder 88
7-10 Symbol finder 89

8-1 Encoding graph 92
8-2 Simulator 95

List of Tables

2.1 High level language usage 8

2.2 Speed changes for caches 18

2.3 Entropy obtained by Bennett. 27

2.4 $\tilde{H}$ obtained by Hammerstrom and Davidson. 30

3.1 Instruction description 35

5.1 Simplified character set for arithmetic coding 47

5.2 Calgary compression corpus 50

5.3 Average number of search loop executions during decoding 51

5.4 Miss distribution for the improved decoding algorithm 54

5.5 Improved decoding results 55

6.1 Opcode distribution 59

6.2 Distribution of opcode pairs 60

6.3 Register distribution for the static case 61

6.4 Register distribution for the dynamic case 61

6.5 Register distribution 62

6.6 Branch distribution 66

6.7 Program description 70

6.8 Encoded object code results for a fixed model in static case 71

6.9 Encoded object code results for a fixed model in dynamic case 71

6.10 Compression schemes 72

6.11 Encoding results published by Bell et.al. 72

7.1 Static and dynamic branch frequency 79

8.1 Compression results using a zero order model 97
8.2 Compression results using a first order model 97
8.3 Compression results compared to entropy using a zero order model . . . 98
8.4 Compression results compared to entropy using a first order model . . . 98
8.5 Entropy improvement from zero to first order model 99
8.6 Comparison between zero and first order model 99
8.7 Effectiveness 100
8.8 Wasted bits using a zero order model 100
8.9 Wasted bits using a first order model 101

Glossary

AMAT	Average Memory Access Time
BN	Branch Never
CISC	Complex Instruction Set Computer
CPI	Cycles per Instruction
CPU	Central Processing Unit
DISC	Distributed Instruction Set Computer Architecture
DMC	Dynamic Markov Compression
DRAM	Dynamic Random Access Memory
EX	Execution
HLL	High Level Language
IF	Instruction Fetch
LZ	Lempel and Ziv compression algorithm [Ziv & Lempel 1978]
LZB	LZ Bell [Bell 1987]
LZFG	LZ Fiala and Greene [Fiala & Greene 1989]
LZW	Lempel Ziv Welch [Welch 1984]
MEM	Data Memory Access
MIN	Multistage Interconnection Network
PC	Program Counter
PPM	Prediction by Partial Matching
PPMC	PPM using method C [Moffat 1990]
RAM	Random Access Memory
RF	Register Fetch

RISC	Reduced Instruction Set Computer
SPARC	Scalable Processor Architecture
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration
WR	Write Register Result

Chapter 1

Introduction

The main thrust in computer architecture design in the seventies was to reduce the semantic gap between high-level languages and machine language. Complex instructions were designed to denote a more substantial component of the user's intended application. One consequence was that a program composed of complex instructions performed the same task as before, but required less memory traffic during execution. In theory such instruction set also allowed the execution unit to run at peak performance more often. An important goal in high-performance architectures is to keep traffic density low on long and shared interconnections [Stone 1990].

1.1 Architectures

In the eighties RISC (Reduced Instruction Set Computer) technology [Fox *et al.* 1986, Hopkins 1987, Katevenis 1984, Patterson & Séquin 1982] has emerged to simplify computer architecture. Instructions are few and simple and control can often be implemented in hardware, removing the overhead of microcode. Design complexity is reduced.

One problem is that static object code size and dynamic object code size [1] could be

[1] *Static code size* is used to mean the code generated by the compiler, prior to execution, and describes the amount of memory occupied by the compiled program. *Dynamic code size* is used to mean the code executed during program run time. Thus, for example, a loop can occupy very little static code size, but huge dynamic code size (depending on the number of loop executions).

significantly larger than for CISCs (Complex Instruction Set Computers). As a result, more memory is required. This applies to caches, main memory, and secondary memory. In addition, memory-processor bus traffic is increased. Much of the work, in particular the design of an advanced architecture, will aim to minimize dynamic code size, since this will increase execution speed.

The problem of increased traffic is multiplied in shared memory multiprocessors where several processes require access to one memory bus. To accommodate several high-performance processors on a single memory bus, the architecture must reduce the bus demands of each processor.

Today the trend is towards an architecture with the best of both worlds. Multicycle instructions can be incorporated successfully into RISC architecture if they improve overall performance without compromising the basic cycle time. This dissertation attempts to increase the benefit of both design methodologies, more object code compactness and less bus loading than CISC, but more speed than RISC.

The increased traffic, both on- and off-chip, requires drastic changes in computer architecture, more so because of the increasing gap between memory and processor cycle time [Hennessy & Jouppi 1991, Hennessy & Patterson 1990, Klaiber & Levy 1991, Kurian *et al.* 1991].

There are several techniques used to hide memory latency, such as caches both on- and off-chip which certainly reduce the amount of data transferred between memory and processor. But no technique known to the author actually decreases memory-processor bus traffic down to the bounds possible [Hammerstrom & Davidson 1977, Schoepke 1992c] according to information theory. Experiments carried out show very low entropy for higher order Markov sources [Abramson 1963] in static as well as dynamic object code [Bennett 1988, Schoepke 1992c]. Such low entropy requires a compression technique which can encode into less than one bit per character.

While a number of research projects are focused on building scalable multiprocessors (scalable to hundreds or thousands of processors) using microprocessor technology, most of the focus in industry is on building small, bus-based machines that support cache coherency [Hennessy & Jouppi 1991]. To accommodate several high-performance processors on a memory bus, the architecture must reduce the bus demands of each processor. However, as machine organizations gradually succeed in sustaining multiple instructions execution concurrently in each processor, they will demand greater bandwidth from local memory.

The work performed by Wade and Stigall [Wade & Stigall 1975], Hammerstrom and Davidson [Hammerstrom & Davidson 1977], and Bennett [Bennett 1988] already show what can be achieved in instruction design.

1.2 Program Size

Programs get larger and larger and the demand for large memory continues to increase [Hennessy & Patterson 1990]. The tendency to use larger memory systems reduces speed as well. The program used requires memory on disk before loading, needs space on the bus during loading and main memory during run time, thus wasting space with redundant information. Figure 1-1 gives us an example of a typical memory hier-

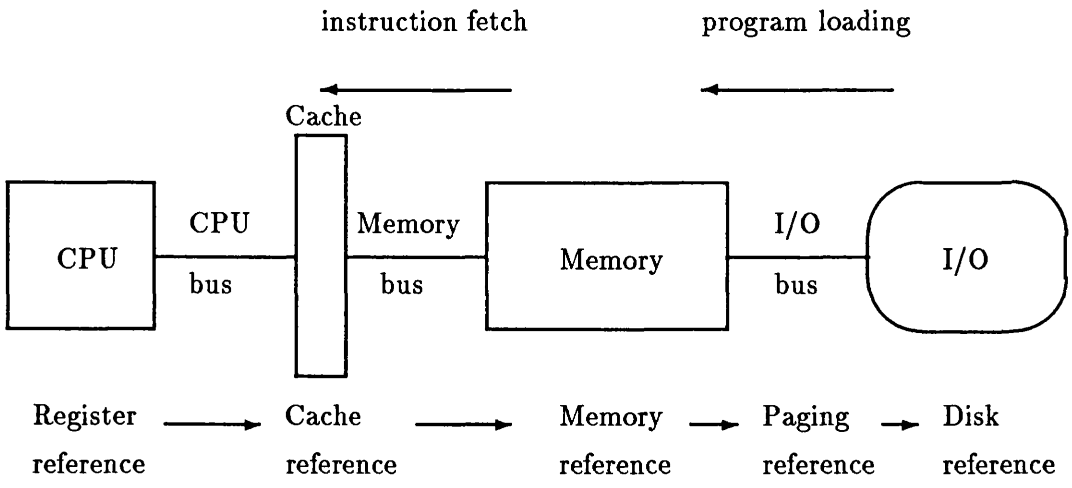

Figure 1-1: Typical memory hierarchy with instruction and reference paths.

archy and the instruction path as well as the memory reference path.

Measurements by Bennett [Bennett 1988] have shown that most of the time people sit in huge programming environments, using commands with small execution times.

Therefore loading times are much more important than is usually appreciated. The memory used by the object code is also in competition with the data memory needed by the average program.

1.3 Memory Latency

Main memory satisfies the demands for caches and vector units as well as serving as the I/O interface. Unlike caches, performance measures of main memory emphasize both latency and bandwidth whereas main memory latency (which affects the cache miss penalty [Hennessy & Patterson 1990]) is the primary concern for caches. A 16MHz PC for example with only one wait state in its memory is slower than another PC that runs at 12MHz with no wait state. Even if the wait state penalty may be only one cycle, it occurs for every memory reference. Techniques that can cope with the large latency of memory accesses [Gupta *et al.* 1991] are essential for achieving high processor throughput. Coherent caches allow shared read-write data to be cached and significantly reduce the memory latency seen by the processor.

Prefetching techniques hide latency by bringing data close to the processor before it is actually needed. These techniques provide performance improvement, but were found to be very application dependent [Gupta *et al.* 1991]. Furthermore, prefetching instructions increases the number of instructions transferred between memory and processor and can end up in prefetching instructions that will never be executed.

1.4 Memory–Processor Connection

Researchers already talk about processors with 1000MIPS and their associated problems. Available bandwidth and latency, not computational speed, will be the main constraint to increase processor performance in the future.

For the 80386 or RISC architectures like SPARC, RISC I or RISC II, where only a single memory port is used for instructions and data access, compact code is even more important. One memory port means that only one access may be in process at any time. The SPARC and 80386 processor for example are both based on the load and store von Neumann architecture. Hence, both can be significantly limited by the memory bandwidth. Assuming a cache hit on SPARC no penalty is incurred. For a cache miss, however, the processor stalls twelve cycles while the line is loaded into the cache. Many cycles are potentially wasted for data transfer operations between the main memory and the CPU internal registers. The data traffic is also competing against the instruction flow from main memory to the CPU, because only a single memory port is provided. Recent work by Ousterhout [Ousterhout 1990] shows that programs run slower on fast machines than the raw speed up would indicate. He believes that low memory bandwidth is one of the reasons for these results.

Introducing caches improve performance significantly, but cache misses are costly. Consider the Fairchild Clipper which has a four kbyte instruction and a four kbyte data cache. The average cache access time is 90nsec, the average memory access time is 400nsec. Assuming a hit rate of 90% the average access time would be 130nsec [2]. This is an increase of nearly 45% in access time compared with a 100% cache hit rate.

Branch prediction strategies are used to improve system performance after a branch has occurred through prefetching instructions at the address concerned. One problem with branch prediction strategies is that unnecessary memory accesses for instructions can be made which will never be executed. This can drastically reduce the amount of memory bandwidth available to the processor. An investigation into this problem has been published by Kaeli and Emma [Kaeli & Emma 1991]. They concluded that branch history tables are an effective approach to reduce latencies that arise looking at taken branches.

1.5 Networking

In networked systems the problem of large object code gets even worse, and swapping and paging across the network seems to be the biggest memory problem for diskless stations. Networks, built with server and several diskless stations, are most common

[2]The formula for the average cycle time t_{eff} is $t_{eff} = t_{cache} + (1-h)t_{main}$ where h is the probability of a cache hit and the time t_{cache} and t_{main} are the respective cycle times of cache and main memory. The quantity $(1-h)$ is the probability of a cache miss.

these days. Serving several Mbyte of code to many diskless stations is not an easy task: disk I/O and transfer through the network account for most of the time spent for executing a short command of up to ten seconds execution time [Bennett 1988]. Loading times, especially across the network, are a far bigger problem than often expected. Object code is a major factor for bus traffic.

1.6 Information Theory

According to information theory, object code compression by more than factor ten is possible considering a fifth order model using eight bit long symbols on SPARC [Schoepke 1992c]. The amount of memory used to store information needed for encoding and decoding is related directly to the order of the model used. Using models of order ten brings the entropy down to 0.06 bit per symbol, but the amount of memory necessary to store the context information is tremendous.

1.7 Summary

For RISCs, the amount of memory which is occupied by object code has increased. Furthermore, dynamic code size increases and therefore memory-processor bus traffic increases. Drastic changes in the architecture are required to deal with this demand. The solution I am looking for is a compromise of the compression ratio achievable with high order Markov sources (such as order ten) and the amount of memory acceptable to store the necessary information.

Chapter 2

Background Theory

In this chapter background information is provided on usage of high level languages (HLLs), instruction sets, strategies for architecture improvements, and information theory. [1] "In any design work it is important to get a feel for the overall structure of languages." [Bennett 1988]

2.1 Analysis of High Level Language Usage

Analysis of HLL usage is essential to get information about instruction usage and the related problems. Several researchers examined language and instruction usage on different machines for various programming languages to find the best representation of a HLL program for execution. From the code size point of view, static measurements of programs are interesting. From the performance point of view, dynamic measurements of programs are generally more interesting than static measurements, but they are also more difficult to collect. Consequently there are fewer of them.

Weiker published a summary of sixteen analyses of high-level language usage gathered by several researchers [Weiker 1984]. He made an attempt to construct a synthetic benchmark program "Dhrystone" based on these recent statistics, particularly in the area of systems programming.

[1]The term *instruction set* refers to the interface presented to the compiler writer by the architecture. The term *architecture* means the complete hardware of the computer, supporting the instruction set.

Weiker gave a brief characterization of all sixteen different data collections and several tables, which summarized the static statistics as well as dynamic statistics. Table 2.1 shows the variety of collected data for several HLL constructs for static as well as dynamic case. In all cases the three constructs of assignments, procedure calls, and conditional branches dominated.

Language Usage

	assignments	**calls**	**if**	**other**
static	33.7-54.0%	12.0-40.3%	8.8-21.1%	7.9%-31.9%
dynamic	41.9-67.0%	4.0-32.8%	7.7-36.0%	1.2%-21.8%

Table 2.1: High level language usage published by Weiker. Information is given about the distribution of different language statements.

In several other tables Weiker shows types of assignments, the distribution of the number of parameters in *calls*, number and type of operators and the locality of variables. The average number of parameters in *calls* varies from 0.9 [Cook & Lee 1982] to 2.1 [DePrycker 1982].

Weiker's work is an excellent summary – as has been shown by the great amount of interest in his Dhrystone Benchmark. Dhrystone is also used in this dissertation to measure the improvement achieved on reduced memory-processor bus traffic. It is worth looking in more detail at some of the work he summarizes and at more recent work.

Knuth was the first to publish work of analysis of the use of existing machines to provide data for future design for computer architecture [Knuth 1971]. He examined a range of FORTRAN programs from industry and academia. He discovered that the average expression has only two operands, indicating that support for complex expression evaluation is perhaps unjustified. In particular he noted that a large number of expressions were of the form $x + 1$ (40%) or y^2 (39%).

About 4% of the statements were *do* loops of which most were quite short involving only one or two statements. Only 13% had more than five statements.

Knuth also distinguished between static and dynamic statistics, showing for the first time that optimization for program size and program speed require different concepts. When he examined twenty-four special programs, he showed how the relative frequency of statement types changes (although not greatly) when the counts are dynamic instead of static. Overall, Knuth's study showed that a small number of basic patterns account for most of the programming constructions in use and that programs generally are very simple.

Alexander and Wortman studied the static and dynamic characteristics of programs written in the language XPL [Alexander & Wortman 1975]. The nineteen XPL programs they examined included compilers written by undergraduate and graduate students as well as two principal components of the XPL system. For numeric constants they obtained the interesting result that 56% of all numeric constants could be represented using four bits, and 98% could be represented using 12 bits or less. Many of the numeric constants greater than 2^{10} were, in fact, masks.

Tanenbaum made an empirical study of more than 10,000 lines of program text to propose an advanced machine architecture specifically designed for structured programs [Tanenbaum 1978]. The programs examined for his research were all written by the faculty and graduate students of a computer science group. Although this work was done much later and concerned a different language, in general they got the same result as Knuth during his FORTRAN study, i.e. programs tend to be very simple.

Patterson and Séquin looked at the frequency of classes of variables in high-level language programs (written in the programming languages C and Pascal) during their RISC project [Patterson & Séquin 1982]. Their most important observation was that integer constants appeared almost as frequently as arrays or structures. More than 80% of the scalars were local variables and more than 90% of the arrays or structures were global variables.

Bennett describes static frequencies of various BCPL statements [Bennett 1988]. In line with other researchers' results his figures are dominated by assignment, procedure call and conditionals, which together account for 82.7% of all statements.

Summary

Different analyses of language usage show that programs of a wide variety of types are generally very simple. During procedure calls only a few parameters are passed and immediates and branch displacements use only a few bits to represent their value. These analyses suggest a great deal of redundancy in computer programs.

2.2 Analysis of Instruction Sets

Looking at high level constructs is not always a sufficient way of analysing language usage since the representation of statements with instructions differs widely. Assigning a constant to a simple variable is inevitably less verbose than assigning a result of a function call to a member of a structure. To get a feel of instruction set usage it is helpful to look at compiled code rather than a HLL.

Alexander and Wortman also studied instruction set usage during their XPL program analysis [Alexander & Wortman 1975]. They found a dominance of the *load* instruction: more than one out of four compiled or executed instruction is a *load*. *Branch and condition* and *store* instructions are both about as frequent, in the range 10-15%.

They also looked at instruction pairs and triples. They found that thirty out of forty-five distinct instructions emitted by the XPL compiler had fewer than four different instructions as possible successors. Therefore these instructions contain more information than just their own function, since they also constrain the possible instructions that might be executed next.

An analysis of branch instructions showed that 55% of the executed branches and 36% of the compiled branches were unconditional. It is interesting to note that over half the branches were no more than 128 bytes away from the location of the branch instruction. That means branch instruction carry a lot of redundant information in them.

Sweet and Sandman, using an analysis of the Mesa byte-stream instruction set, describe the refinement of the instruction set [Sweet & Sandman 1982]. They provide a formalization of the statistics required and the method to be used. Under their scheme to reduce the static size of compiled code, they propose a five stage design process:

1. *Normalize the object code:*

   Many of the instructions in existing object code were special versions of generic instructions dealing with common cases. These were replaced by their generic instructions, to eliminate pre-supposition about what were "good" instructions. For example *jump not zero* was replaced by *load zero* followed by *jump not equal.*

2. *Collect statistics by pattern matching:*
   2.5 million bytes of code were analysed to find particular statistical information.
   - *Static opcode frequency*: count the number of occurrences of each opcode;
   - *Operand values*: for each opcode, get a histogram of operand values;
   - *Popular opcode pairs*: for each opcode, get a histogram of the set of next opcodes in the code sequences; this includes *opcode predecessors* and *opcode successors*; these are the same items of information, but different representations are more useful at different times.

3. *Propose new instructions:*
   The statistics gathered were used to suggest new instructions through combining opcodes, opcode and argument, or arguments. The instruction pair *LI 0, LI 0* (Load Immediate Zero) for example led to a new instruction *LID0* (Load Immediate Double Zero).

4. *Peephole optimization:*
   Convert to new opcodes by peephole optimization.

5. Repeat steps 2 through 4 until you have enough instructions.

Normalization is an important concept in refining instruction sets. The existing instruction set utilized 240 out of a possible 256 instructions. Sweet and Sandman's normalized instruction set had 100 generic instructions, leaving scope for 156 new instructions.

For static frequencies Sweet and Sandman obtained the result that *load immediate* and *load local variable* were the most common instructions with 16.90% and 12.68% respectively. Over 6% of the instructions were procedure calls. In fact, only six instructions account for more than 50% of the opcodes. For *load immediate* instructions they found that 45.83% of arguments are zero and 14.01% are unity.

Cook describes a static analysis of the instructions used to implement the system software on the Lilith computer, including editors, document processors, window packages and other modules [Cook 1990]. The analyzed software was from 180 modules with 2236 procedures, comprised of 146293 instructions. All programming on Lilith was done in Modula-2.

Cook obtained the result that 20 out of 256 possible opcodes represent 50% of the usage of all instructions. Overall Cook got comparable results as Sweet and Sandman [Sweet & Sandman 1982] during their analysis about the Mesa instruction set.

During static analysis Bennett obtained similar results for BCPL programs namely that only a few instructions built most of the compiled code [Bennett 1988].

A recent analysis of MIPS and SPARC instruction set utilization using the SPEC benchmarks is given by Cmelik et.al. [Cmelik *et al.* 1991]. Although no details are given for single instruction or register usage, the results give a good overview on how instruction categories are used. In particular, their integer benchmark indicates that between 20.26% and 26.51% of all SPARC instructions measured are control transfer instructions (depending on the benchmark program used).

Summary

Different analyses of instruction set usage show that most of the compiled code is built with only a few instructions, far less than the number of instructions provided by the architecture. New instructions built from frequently used opcodes or opcode pairs can reduce program size. The analyses shown (undertaken by several researchers) demonstrate the simplicity of generated instruction streams.

2.3 Computer Architectures

Several techniques are known that can hide memory latency. Instruction sequencing, scheduling, and pipelining are only a few of them. Hiding memory latency is one of the main concerns of computer architects as the gap between memory and processor cycle times is still widening [Hennessy & Patterson 1990]. Caches can only alleviate this problem as they have to be both, large and on-chip. Furthermore, the cache memory cycle time increases for larger cache size due to additional decode logic and loading of the bus (both internal and external to the memory chip) [Krick & Dollas 1991]. Certain memory organizations are able to reduce latency times, but not the amount of data transferred between memory and processor. Some can even increase traffic, such as prefetch techniques.

2.3.1 RISC Architecture

Patterson and Sèquin [Patterson & Sèquin 1982] describe the Reduced Instruction Set Computer Project. The purpose of their project was to explore alternatives to the general trend towards architectural complexity. They expected a reduction in design time and design errors and much faster execution time for individual instructions. Patterson's and Sèquin's project is one of the research projects that lead the way to the RISC architecture.

While there are many variations on the theme, most RISC processors have the following attributes in common [Colwell *et al.* 1985]:

Fewer instruction

Studies in the early days of RISC showed that most compilers used only 30% of the instruction sets provided by CISCs, such as DEC's VAX. RISC processors attempt to implement only that 30%, allowing the chip to be smaller, cheaper, and (in theory) faster.

Fewer instruction formats

RISCs generally have only a few instruction formats to maintain simplicity. Usually, all instructions are the same length such as in SPARC. Furthermore, on most processors instructions and data words are of the same size. For SPARC this means it can take more than one instruction to load a constant of more than 13 bits as the immediate operand is only 13 bits long. However, as most constants in computer programs are small numbers, a single *add* or *or* instruction suffices in most cases.

Dummy register

Most RISCs have a dummy register, a register with the hardwired value zero. Such a register is a convenient way to turn an instruction with two sources into an instruction with a single source without complicating the instruction set.

Load and store architecture

In most cases load and store instructions are the only instructions to access memory. All other instructions effect only the on-chip registers. This rule is violated on SPARC by allowing atomic load and store such as the swap instruction.

Limited addressing modes

Most RISCs have only simple addressing modes (indirect through a register), doing all address calculations in registers. However, implemented chips such as SPARC, allow also register-plus-displacement modes and some processors support a third addressing mode through forming a memory address by adding the contents of two internal registers.

Single cycle execution

Most RISC instructions are simple enough to require only one cycle. On some architectures multiple cycles are required by instructions such as *load* or *store*.

Delayed control transfers

To avoid wasted time, most RISCs allow an additional instruction to be inserted into the "delay slot" after the control transfer instruction before the branch is actually taken. Some processors such as SPARC can *annul* [2] this instruction.

2.3.2 Shared Memory Architecture

Shared memory architecture accomplishes interprocessor coordination by providing a global, shared memory that each processor can address. Commercial shared memory

[2]See section 3.1.

architectures were introduced during the 1980s. A typical first generation system was the Balance 8000 from Sequent Corporation, offered in 1984 with two to twelve National 32032 microprocessors. The 8000 was updated two years later by the 21000, offering up to 30 microprocessors. Two further commercial shared memory architectures are the Flex/32 from Flexible Corporation and the Multimax from Encore Computer. These architectures involved multiple general-purpose processors, sharing memory. Processes communicate through shared variables in memory. However, synchronization must be available to coordinate processes.

Shared memory computers avoid some of the problems encountered by message passing architectures, but the problems of data access synchronization and cache coherency must be solved. Typically, each processor in a shared memory architecture has also a local memory used as a cache. Multiple copies of the same shared memory data may therefore exist in various processors' caches at any given time. Maintaining a consistent version of such data is the cache *coherency* problem.

To maintain cache coherency a special protocol, called *snooping*, can be used. Each cache controller monitors, or *snoops*, on the bus to determine whether they have a shared block of data. This technique is popular on shared memory systems as it can use a pre-existing connection: the memory bus.

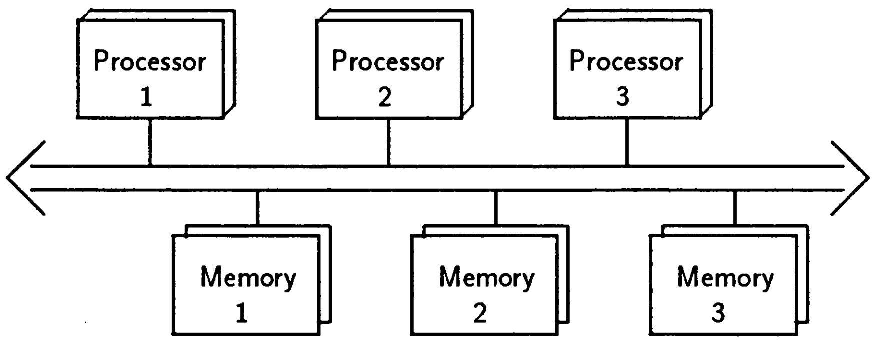

Figure 2-1: Multiprocessor Bus Interconnection

There are a number of ways of connecting multiple processors to shared memory [Duncan 1990]. *Bus interconnection* (Figure 2-1) offers the simplest way to give multiple processors access to shared memory. A single time-shared bus efficiently accommodates a moderate number of processors, limited by the fact that only one processor may use the bus at any given time.

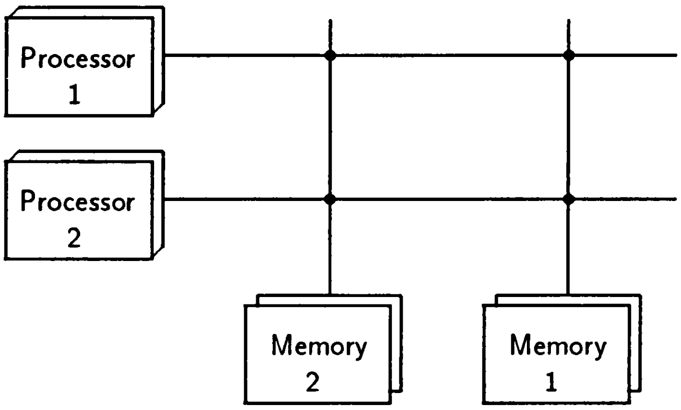

Figure 2-2: Multiprocessor Crossbar Interconnection

Crossbar interconnection (Figure 2-2) uses a crossbar switch of n^2 crosspoints to connect n processors to n memories. Processors may contend for access to a memory location, but crossbars prevent contention for communication links by providing a dedicated pathway between each possible processor/memory pairing.

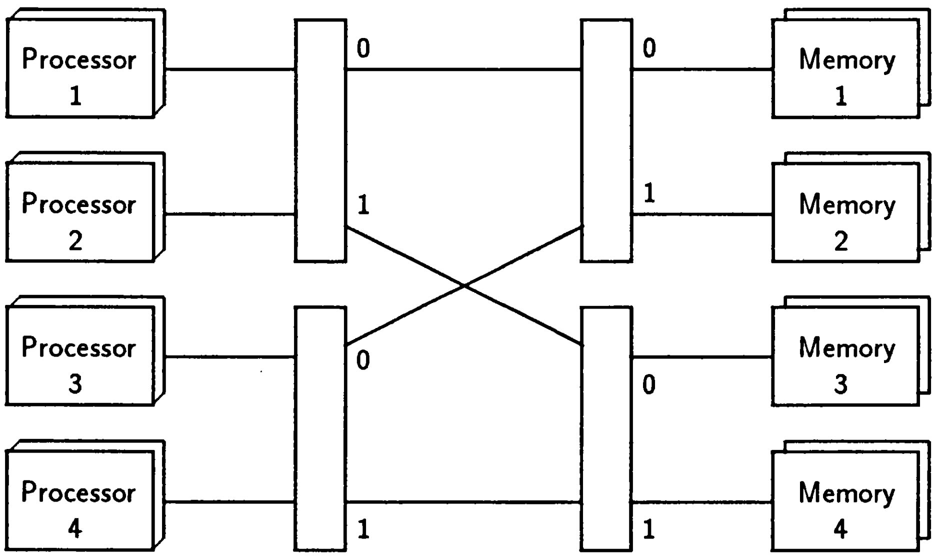

Figure 2-3: Multistage Interconnection Network

The *multistage interconnection network* (MIN) shown in Figure 2-3 strikes a compromise between the price/performance alternatives offered by crossbars and buses. An $n * n$ MIN connects n processors to n memories by deploying multiple stages or banks to switches in the interconnection network pathway. A processor making a memory access request specifies the desired destination and pathway by issuing a bit-value that contains a control bit for each stage. Multiprocessor performance heavely depends on the performance of the whole system when sharing data.

2.3.3 Improving Main Memory Performance

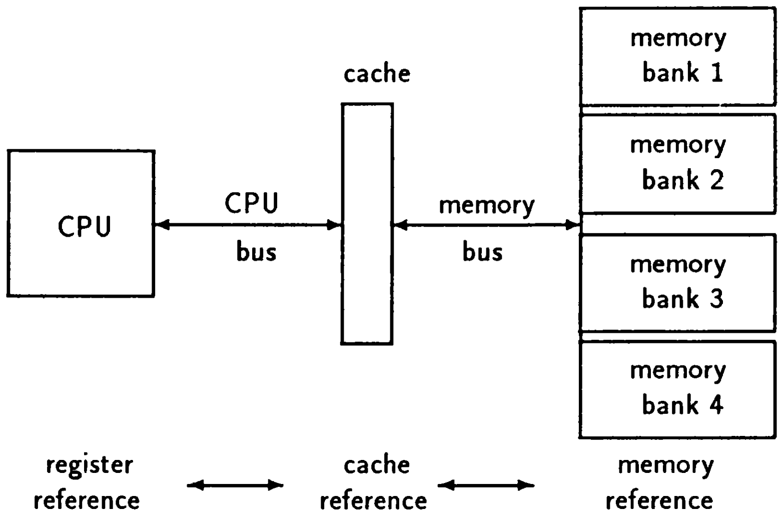

Figure 2-4: Interleaved memory organization with four memory banks.

Main memory can be organized in banks (Figure 2-4) to give access to multiple words at a time rather than single words. The banks are one word wide so that cache as well as bus width need no changes. Organized in banks, main memory allows one clock cycle for each write (provided the writes are not destined to the same memory bank). This "mapping" of addresses to banks affects the memory behaviour as the memory address is interleaved.

The original motivation was to interleave sequencial memory accesses and to allow multiple independent accesses. This method, however, does not decrease the number of bytes to be transferred between main memory and processor. The memory bus is still the main bottleneck in the system.

2.3.4 Cache Influence

One of the most important hardware techniques used to improve performance during the past decade has been caching; a technique which relies on using more hierarchical memory to achieve higher performance. Many researchers have investigated caches, their performance and improvements, and influence on different architectures, for example [Eickemeyer & Patel 1988, Farrens & Pleszkun 1989].

An instruction cache greatly speeds the instruction-sequencing requirements of the instruction fetch unit. A cache hit must occur to improve the apparent memory cycle time. To attain a peak execution rate of one instruction per cycle, the cache memory must have a cycle time less than or equal to the instruction fetch time. Memory access time, however, has to be "reasonable" so cache misses are not too costly. Second level caches between the primary cache and main memory can be used to reduce cache miss time.

Speed Changes			
Hit Rate	**Hit Time**	**Miss Time**	**Total Time**
percent	*percent*	*percent*	*percent*
100%	100%	0%	100%
99%	99%	26%	125%
98%	98%	52%	150%
95%	95%	130%	225%

Table 2.2: Speed changes as hit rate varies for the SPARCstation 2 with 26 cycle miss costs. The hit rate reduces from 100% to 95%. The effect on hit time, miss time and total time is shown.

Although the cache hit ratio is largely affected by cache size, increase in size must also be weighed against longer access time and higher costs. For a given memory technology, the cache memory cycle time increases for larger cache size due to additional decode logic and loading on the bus. Cache efficiency depends heavely on the cache hit rate. Table 2.2 provides information about speed changes for the SPARCstation 2 with a cache miss cost of 26 cycles [Cockcroft 1991] as the cache hit rate varies. Although a hit rate of 95% sounds high, there is a dramatic increase in execution time compared with a 100% hit rate.

2.4 Instruction Execution

Instruction sequencing is one of the fundamentals for von Neumann architectures, but it is complicated by interdependencies between instructions. High performance systems use a number of strategies to address these interdependencies in order to improve system performance. Krick and Dollas [Krick & Dollas 1991] discuss several aspects of instruction sequencing such as memory bandwidth, instruction buffers, and caches.

The memory address that contains the next instruction to be executed must be known before the instruction can be fetched and executed. A processor can achieve peak performance only when it does not have to wait for memory to provide the instruction required for execution. During the execution phase of each instruction, the processor determines the memory location of the next instruction to be executed. Instructions such as conditional branches can affect the address of the next instruction and the memory response could affect the availability of an instruction for execution. "Delayed jumps" as used in SPARC or R3000 offer the opportunity to insert an additional instruction in a branch *slot* which will be executed before the branch will be taken. Prefetching of instructions into a buffer and caches are two solutions which improve system performance [Hennessy & Jouppi 1991].

The past decade has seen renewed interest in instruction sequencing. Designers have proposed a variety of hardware and software approaches such as branch prediction strategies and instruction-scheduling techniques to further improve system performance.

Instruction scheduling or *pipeline scheduling* was first used in the 1960s and became an area of major interest in the 1980s, as pipelined machines became more widespread. The CDC 6600, delivered in 1964 by Control Data, was one of the first machines using an instruction scheduler [Thorlin 1967]. Rather than allowing a pipeline to stall, a compiler could try to avoid these stalls by rearranging the code sequence generated. Measurements undertaken with the DLX pipeline show that 54% of loads result in a pipeline stall when using the gcc compiler. Using instruction scheduling the number decreases to 31% [Hennessy & Patterson 1990]. These pipelining *hazards* occur when, for example, the result of a load instruction is used by the next instruction as a source operand.

Instruction pipelining is a technique which allows the decomposition of instruction execution into a series of autonomous stages which can be executed independently. The first general-purpose pipelined machine is considered to be the IBM 7030. The more recently Mips R3000 architecture is implemented with a five stage pipeline. Figure 2-5 shows an example for a simple pipeline structure with three pipe stages.

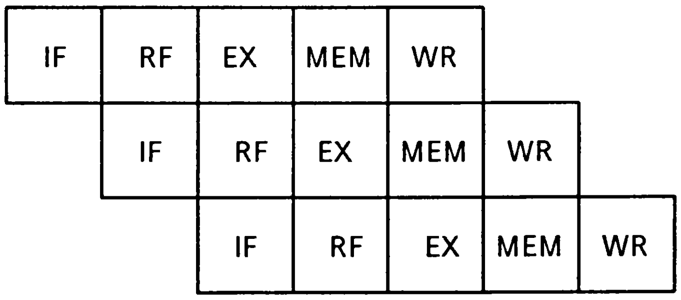

Figure 2-5: A simple machine pipeline with three pipe stages. (IF is used to mean instruction fetch, RF register fetch, EX execution, MEM data memory access, and WR write register result.)

In pipelining systems, subsequent instructions can be requested before the execution of the previous instruction has been completed. The problem is that conditional branch instructions, however, can alter the request of subsequent instructions in a pipelined system.

Pipelining improves the throughput of a machine by exploiting instruction-level parallelism without changing the basic cycle time. In fact, the execution time of each individual instruction can increase slightly due to overhead in the pipeline control. Instruction-level parallelism is available when instructions in a sequence are independent and can thus be executed in parallel by overlapping [Hennessy & Jouppi 1991]. The machine attains its performance benefits by increasing the number of pipeline stages in the processor and keeping all stages busy.

However deeply pipelined processors have relatively low issue rates [3] due to dependencies between instructions [Farrens & Pleszkun 1991]. Multiple instruction issue is offered by machines such as Apollo DN10000 and Intel i860.

Super pipelining increases performance even further. Using this technique, one step in the pipeline can be done in less than a machine cycle. Figure 2-6 shows an example for a simple pipeline structure with three pipe stages. The Mips R4000 architecture is implemented with a eight stage super pipeline using half the cycle time internally than externally. For R4000 super pipelining has been chosen against superscalar because it need less implementation logic on-chip [Gänsheimer & Reisch 1991].

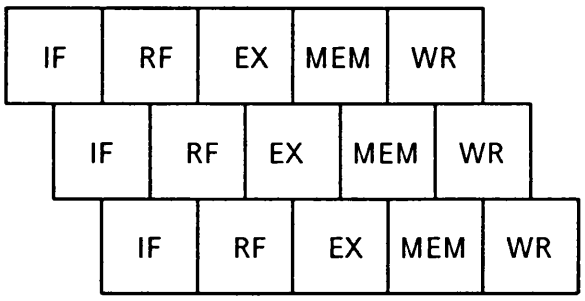

Figure 2-6: A simple machine super pipeline with three pipe stages.

2.5 Architectural Solutions

2.5.1 Superscalar Designs

Superscalar designs are architectures that can execute more than one instruction per cycle. Hardware can be used to identify independent instructions which can be executed in parallel. Two major studies of scalar oriented programs have shown the practical level of exploitable parallelism to be around two operations per clock [Hennessy & Jouppi 1991]. All superscalar machines built to date restrict the combination of instructions that can be issued in one clock. One implementation is the IBM RS/6000 [Grohoski 1990]. Ini-

[3] The process of letting an instruction move from the instruction decode stage into the execution stage of the pipeline is called instruction *issue*; the instruction that has made this step is said to have *issued*.

tial results from this work are very promising, and the commercial implementation and availability of these processors demonstrate a mature technology. The Intel 960 CA and Tandem Cyclone are two examples of superscalar machines with complex instruction sets. Figure 2-7 shows an example for a simple superscalar structure with two pipe stages.

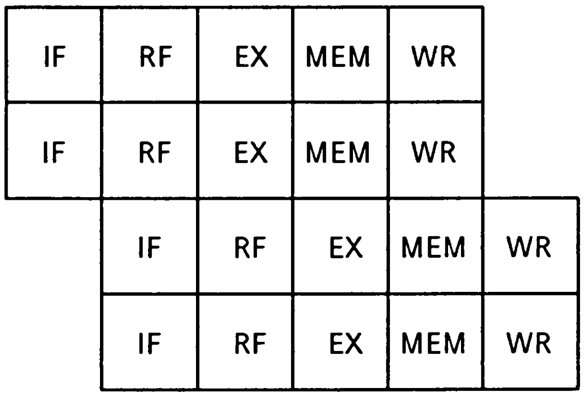

Figure 2-7: A simple superscalar structure with two pipe stages.

The limiting factor in increasing the instruction issue rate for a superscalar machine is probably the difficulty in fetching, decoding, and issuing an ever-larger number of instructions in the same clock cycle. One question yet to be resolved is whether the main memory organization of such processors can follow conventional designs, or whether the increased traffic requires drastic changes.

2.5.2 Primitive Based Architectures

Fritsch et.al. [Fritsch *et al.* 1990] maintain that the only way to raise performance if code fetch bandwidth is limited, is to increase code density. For this purpose and to reduce the cycle time required by any conventional RISC architecture, they define a family of *Primitive Based Architectures*, characterized by uniformity of coding and fast parallel decoding. The difference between instructions and such *primitives* is that an instruction completely defines an operation, while a *primitive* of order n expresses the n elemental actions to be carried out, on arguments possibly defined by preceding actions.

For a primitive set for order two, for which they obtained the shortest code length, Fritsch et.al. evaluated primitive formats and a primitive set including conditional and unconditional branching, operations, memory access and other features like long literal handling and stack operations. Assuming the same register-window configuration as the Berkeley RISC-II [Katevenis 1984] and four stage pipeline in the processor, Fritsch et.al. achieved shorter code length and faster execution than the RISC-II machine during simulation.

2.5.3 Very Long Instruction Words

Another approach to increase concurrency uses *Very Long Instruction Words* (VLIWs) [Fisher 1987]. Memory words contain more than one instruction, so each memory reference fetches multiple instructions. This architecture relies heavily on sophisticated compilers to generate the code required to use machine resources efficiently. A major impediment to such architectures is the conditional branch. When a conditional branch is pending, scheduling useful computations is difficult until the execution path becomes known.

In contrast to superscalar architectures, in VLIW architectures the compiler has complete responsibility for creating a package of instructions that can be issued simultaneously and the hardware does not makes the decisions about multiple issue. TRACE – built by Multiflow Computer – is a VLIW architecture [Colwell *et al.* 1987].

2.5.4 Micro-Architectural Parallelism

Hwu and Chang [Hwu & Chang 1988] evaluated *micro-architectural parallelism*, including multiple instructions issued per cycle, multiple result distribution buses, multiple execution units and pipelined execution units. All of their architecture variations had a split register organization with 32 integer and 32 floating point registers. Code was generated with a prepass code generation strategy, which performed instruction scheduling before register allocation. Using the Livermore Loop and Linpack subroutines as benchmarks, Hwu and Chang concluded that, when used together, multiple instruction issue and pipelined execution units produced a speedup greater than the sum of speedups of each taken separately. They also found that issuing more than two instructions per cycle produced little additional speedup.

Higher instruction execution rate also led to an increase in bus traffic as more instructions were fetched from memory per processor cycle.

2.5.5 Distributed Instruction Set Computer Architecture

The *Distributed Instruction Set Architecture* (DISC) employs a new parallel instruction set and a distributed control mechanism to explore fine-grained parallel processing in a multiple-functional-unit system. Multiple instructions are executed in parallel and/or out of order at the highest speed of n instructions per cycle, where n is the number of functional units. Wang and Wu [Wang & Wu 1991] developed a hardware system and studied the performance level, the effect on program size and the hardware utilization. Their simulation results show that a DISC system incorporating 16 functional units can run 7.7 times faster than a single-functional-unit DISC system. DISC presents three major contributions in the domain of fine-grained multiprocessing:

1. Fast multiple instruction issuing mechanism: no decoding work is needed prior to issuing an instruction. Multiple instructions are fetched from memory and issued directly to multiple functional units.

2. Parallel and/or out-of-order execution: data dependency among functional units is maintained in a distributed manner. No hardware is required to coordinate multiple instruction executions. This minimizes the inter-unit communication and speeds up the overall execution rate.

3. Software dataflow: the post compiling idea is to tag each instruction with its data tag. This is equivalent to generating the data token in software and combining it with the instruction token. It pioneers a software dataflow control scheme for multiple-functional-unit systems.

In contrast to dense instruction sets, DISC is based on instruction independency. The results provided in Section 6.3 question these independencies. However, executing more than one instruction per cycle also implies that more instructions per cycle have to be fetched from memory. Assuming an execution of two instructions per cycle, bus traffic at least doubles.

Summary

Bus demands increases not only for shared memory multiprocessors. In addition for processors which are able to execute more than one instruction per cycle the increasing bus traffic requires changes in the architecture to supply instructions to the execution unit fast enough.

2.6 Theoretical Analysis of Instruction Sets

The concepts of information theory and its mathematical analysis are established in Shannon's source coding theorem [Shannon 1948]. Many textbooks now provide an introduction to information theory, for example [Abramson 1963]. A number of attempts have been made to use information theory in instruction set design [Bennett 1988]. Modelling the structure of instruction sets has been used to estimate the most compact instruction set [Flynn & Hoevel 1984]. Entropy has been considered to examine static or dynamic code size [Bennett 1988, Wade & Stigall 1975] and redundancy in addressing [Hammerstrom & Davidson 1977].

Analysis involving the entropy of instruction streams offer far more hope. Entropy [Abramson 1963, Thomas 1991] is the key to compression and understanding entropy is vital to understanding compression.

2.6.1 Overview

A *zero-memory source* is given by a sequence of symbols from an alphabet S

$$S = \{s_1, s_2, \ldots, s_q\} \tag{2.1}$$

where the occurrence of a symbol is independent of previous symbols. Such an information source is described completely by the source alphabet S and the probabilities with which the symbol occurs:

$$P(s_1), P(s_2), \ldots, P(s_q).$$

If symbol s_i occurs, one obtains an amount of information equal to

$$I(s_i) = -log_2 P(s_i)\ bits \tag{2.2}$$

according to information theory [Shannon 1948]. The average amount of information obtained per symbol is thus

$$H(S) = \sum_{i=1}^{q} P(s_i) I(s_i)\ bits$$

where $\sum_{i=1}^{q}$ means the summation over all q symbols. This quantity, $H(S)$, the average amount of information per source symbol, is called the *entropy* of the zero-memory source. When a stream of symbols is encoded so it can be expressed in the fewest possible bits per symbol, it is said to be encoded optimally. An optimal encoding of a stream of symbols uses $H(S)$ bits.

To get more information about the source S one must deal with blocks of symbols rather than individual symbols. In a more general type of information source symbol s_i is dependent on one or more preceding symbols. This leads to a generalization of the zero-memory source and to the m^{th}-order Markov information source with a set of conditional probabilities.

For an m^{th}-order Markov source, the probability that a given symbol appears depends on the m preceding symbols. Considering the preceding m symbols,

$$(s_{j1}, s_{j2}, \ldots, s_{jm})$$

as a single *state* then the conditional probability of the next symbol s_i is

$$P(s_i / s_{j1}, s_{j2}, \ldots, s_{jm}).$$

The information obtained if s_i occurs while one is in state $(s_{j1}, s_{j2}, \ldots, s_{jm})$ is

$$I(s_i/s_{j1}, s_{j2}, \ldots, s_{jm}) = -log_2 P(s_i/s_{j1}, s_{j2}, \ldots, s_{jm}) \,. \tag{2.3}$$

The average amount of information or *entropy* of the m^{th}-order Markov source S:

$$H(S) = -\sum_{S^{m+1}} P(s_{j1}, s_{j2}, \ldots, s_{jm}, s_i) log_2 P(s_i/s_{j1}, s_{j2}, \ldots, s_{jm}) \tag{2.4}$$

where S^{m+1} is the $(m+1)^{th}$ extension of the zero-memory source S. The probability of state $(s_{j1}, s_{j2}, \ldots, s_{jm}, s_i)$ is $P(s_{j1}, s_{j2}, \ldots, s_{jm}, s_i)$:

$$P(s_{j1}, s_{j2}, \ldots, s_{jm}, s_i) = P(s_i/s_{j1}, s_{j2}, \ldots, s_{jm}) P(s_{j1}, s_{j2}, \ldots, s_{jm}). \tag{2.5}$$

A computer program is assumed to consist of a stream of symbols, the instructions. Measurements by Bennett [Bennett 1988] have already shown the dependency between opcodes and complete instructions. His entropy observations are shown in Table 2.3, giving the entropy for different order models.

	Entropy	
Order	**Opcodes Only**	**Complete Instructions**
	bits/symbol	*bits/symbol*
0	3.98	3.24
1	2.87	2.64
2	2.35	2.28
3	2.03	2.01
5	1.19	1.26
10	0.13	—
25	0.01	—

Table 2.3: Entropy obtained by Bennett.

The reduction in entropy is nearly 28% from the zero order to the first order process. For fifth order he reaches one bit requirement per symbol and for higher order even less.

2.6.2 Replacement of Repeated Strings

Wade and Stigall used information theory (and the concept of entropy) to attempt to estimate how much space object code should require [Wade & Stigall 1975]. Their approach was to treat instruction streams as a sequence of (independent) symbols. One of the simplest and most common ways to reduce the length of a symbol stream is to identify repeated strings and replace them with a single symbol [Wade & Stigall 1975]. If one assumes the string

$$s_1, s_2 ... s_k$$

occurs with probability P_s and is replaced by the new symbol s' whenever it occurs, then Wade and Stigall show the new value of entropy H' is given by

$$\begin{aligned} H' &= -\frac{P_s}{1-(k-1)P_s} log_2(\frac{P_s}{1-(k-1)P_s}) & (2.6) \\ & -\sum_{i=1}^{k} \frac{P_i - P_s}{1-(k-1)P_s} log_2(\frac{P_i - P_s}{1-(k-1)P_s}) & (2.7) \\ & -\sum_{i=k+1}^{N} \frac{P_i}{1-(k-1)P_s} log_2(\frac{P_i}{1-(k-1)P_s}) & (2.8) \end{aligned}$$

The number of symbols to represent a string is reduced by factor

$$i - (k-1)P_s$$

The resulting change in entropy normalized to the length of the original string is given by

$$H_n - H = (1 - (k-1)P_s)H' - H \ . \qquad (2.9)$$

where H_n is the normalized entropy. Wade and Stigall then considered the values of P_s for which the replacement results in a decrease of entropy. For small P_s they showed that this is given by

$$P_s > e \prod_{i=1}^{k} P_i$$

where e is the base of the natural logarithm. Their result quantifies the idea that a rarely occurring long string of symbols can often be replaced by a new symbol and save memory, while a shorter string must occur often to save memory.

The assumption made by Wade and Stigall that a stream of instructions can be considered as a sequence of independent symbols is very weak as shown by Bennett [Bennett 1988]. His opcode and instruction measurements show high dependency.

2.6.3 Memory Referencing Behaviour

Hammerstrom and Davidson [Hammerstrom & Davidson 1977] used a similar analysis to estimate the information content of a memory referencing stream in the IBM S/360, using information theory. Specifically, they present techniques for analyzing computer addressing architectures and techniques for analyzing the efficiency of the addressing architecture and memory/CPU traffic of existing machines with regards to the information theory boundaries.

To obtain further improvement in memory/CPU bandwidth and CPU addressing efficiency, they suggest looking at higher order memories and accompanying radical changes in CPU architecture and compilation techniques. For the extension to higher order conditional probabilities they find a more useful definition of entropy.

Hammerstrom and Davidson also define the *absolute entropy*

$$H_\infty(S) = \lim_{n\to\infty} H_n(S). \tag{2.10}$$

The calculation of $H_\infty(S)$ for a finite program trace is impossible, since as n approaches the length of the trace, $H_n(S)$ approaches 0 and is no longer meaningful. Therefore, Hammerstrom and Davidson introduced a variable $\tilde{H}$, which is an estimate of $H_\infty(S)$ based on the general structure of, but not on complete knowledge of, the trace. Their results (the estimated entropy, $\tilde{H}$, and the addressing overhead, $\tilde{A}$ [4]) collected on the IBM 360 machine compiled with the Fortran G compiler are shown in Table 2.4. The

[4]The addressing overhead is defined as the amount of bits necessary for the addressing process divided by the number of computational process references (which is built by the number of references minus the sum of instructions without reference bits plus the amount of data references).

Gauss program is a Gaussian elimination on a 14x14 matrix, the Error program is a floating point benchmark for the IBM 360. These results indicate there is a lot of

Program	$\tilde{A}$ *bits/reference*	$\tilde{H}$ *bits/reference*
Gauss	17.2	1.64
Error	10.0	0.01

Table 2.4: $\tilde{H}$ obtained by Hammerstrom and Davidson.

redundancy in instruction streams.

2.7 Summary

Study of the relevant literature shows that memory-processor bandwidth is a serious problem, especially for multi processors with shared memory where the reduction in memory-processor bus traffic promises performance improvement. Furthermore, slow memory latency time can reduce system performance. Treating instructions as higher order Markov sources shows great redundancy in instruction streams. Therefore, reducing redundancy in instruction streams is a plausible strategy to improve system performance.

Chapter 3

A Target Architecture for Experimental Compression

The SPARC architecture has been chosen as the basis on which to implement and simulate the dense architecture developed. Therefore, information is provided about instructions and timings using SPARC.

In 1987, Sun Microsystems announced the Sun-4, the first computer based on the new SPARC (Scalable Processor Architecture) RISC processor. The RISC architecture philosophy evolved from research projects at the University of California at Berkeley and Stanford University in the early 1980s and is already described in Section 2.3.1. Many of their features are part of the SPARC architecture. Like most RISC chips SPARC uses pipelining to allow concurrent execution.

3.1 SPARC Architecture

Features

The SPARC architecture provides the following features: *Simple instructions* as most instructions require only a single arithmetic operation; *few and simple instruction formats* as all instructions are 32 bits wide; *register intensive architecture* as most instructions operate on either two registers or one register and a constant; *a large windowed register* *field*; *delayed control transfer* as the processor always fetches the next instruction after a control transfer; *one cycle execution*; *concurrent floating-point*; and a *co-processor interface.*

Instruction Types

The SPARC instructions fall into six basic categories:

- *Load/store:*
  These instructions are the only instructions that access memory. They use two integer unit registers or an integer unit register and a signed immediate value to calculate the memory address. Integer load and store instructions support halfword, word, and double word accesses. Floating-point and co-processor load and store instructions support word and double word accesses.

- *Arithmetic/logical/shift:*
  These instructions (with one exception) compute a result that is a function of two source operands; they either write the result into a destination register or discard it. The exception is a specialized instruction used to create 32 bit constants in two instructions.

- *Control transfer:*
  Control transfer instructions include jumps, calls, traps and branches. Control transfer is usually delayed, but a special bit can cause the delay instruction (the instruction following the branch instruction) to be *annulled* [1] if the branch is not taken. Branch and call instructions use PC-relative displacement. Jump and link uses a register-indirect displacement. The branch instruction provides a displacement of 8Mbytes. Branch instructions and procedure calls use a fixed displacement field (*disp22* and *disp30* respectively) multiplied by four and added to the program counter to get their destination address. Jump instructions, mostly

[1] On conditional branch instructions the *annul* bit changes the behaviour of the delay instruction. If the annul bit is set and the branch in not taken, the delay instruction is not executed, i.e. it is annulled. If the branch is taken, the annul bit is ignored and the delay instruction is executed.

used to return from subroutines, use a register indirect address, calculated at program run time. A jump and link instruction with destination register 15 can be used as a register indirect call. As the *call* instruction writes its address in *out* register seven, the jump and link instruction (used as return) is able to return to the instruction following the *call* delay instruction.

- *Read/write control register:*
  Read and write instructions are provided to read and write the contents of the various control registers.

- *Floating-point operate:*
  Floating-point operate instructions perform all floating-point calculations, which are *register-to-register* instructions that use the floating-point registers. The result is a function of two source operands.

- *Co-processor operate:*
  Co-processor instructions are defined by the implemented co-processor, if any. The architecture supports 1024 distinct co-processor arithmetic instructions.

3.2 Instruction Description

The SPARC instructions shown in Figure 3-1 can be classified into three different instruction formats, two of which include subformats. The instruction format one is used for the *call* instruction only and contains two different bit fields, the instruction format two is used for *sethi* and branch instructions, floating point and co-processor branches inclusive, and the instruction format three is used for all other instructions.

The terms used are:

- *op* is the opcode bit field which places the instruction into one of the three formats.

- *op2* selects the instruction from the second format.

- *op3* selects the instruction from the third format.

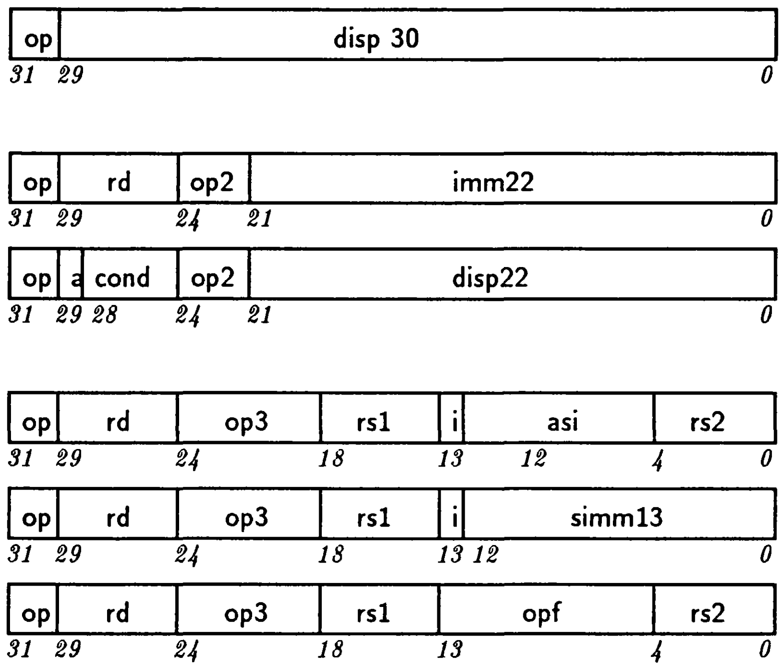

Figure 3-1: Instruction formats for the SPARC instruction set.

- *rd* specifies the destination register for all instruction with the exception of the load instruction where it describes the source register.
- *a* is the annul bit in format two.
- *cond* selects the condition code for format two instructions.
- *imm22* is the 22 bit constant value used by *sethi*.
- *disp22* is the 22 bit displacement value for branch instructions.
- *disp30* is the 30 bit displacement value for call instructions.
- *i* specifies whether the second operand in format three is a register or a constant.
- *asi* is the address space identifier (available to the external system) which distinguishes up to 256 address spaces. In the SPARC architecture *asi* bits define four address spaces: user or supervisor instruction space and user or supervisor data space.

- *rs1* specifies the source register one.
- *rs2* specifies the source register two.
- *simm13* is the 13 bit value used as a second operand if the *i* bit is one.
- *opf* identifies a floating point operate instruction.

In Table 3.1 a short instruction description is given for all instructions mentioned, except for branch instructions which are shown in Table 6.6.

Instruction Description

opcode	*parameter*	*operation*
add	$reg_{rs1}, reg\_or\_imm, reg_{rd}$	add
and	$reg_{rs1}, reg\_or\_imm, reg_{rd}$	and
andcc	$reg_{rs1}, reg\_or\_imm, reg_{rd}$	and and modify icc
call	*label*	call
jmpl	$address, reg_{rd}$	jump and link
ld	$[address], reg_{rd}$	load word
ldsb	$[address], reg_{rd}$	load signed byte
ldub	$[address], reg_{rd}$	load unsigned byte
lduh	$[address], reg_{rd}$	load unsigned halfword
or	$reg_{rs1}, reg\_or\_imm, reg_{rd}$	or
orcc	$reg_{rs1}, reg\_or\_imm, reg_{rd}$	or and modify icc
restore	$reg_{rs1}, reg\_or\_imm, reg_{rd}$	restore caller's window
sethi	$imm22, reg_{rd}$	set high 22 bits of register
sll	$reg_{rs1}, reg\_or\_imm, reg_{rd}$	shift left logical
srl	$reg_{rs1}, reg\_or\_imm, reg_{rd}$	shift right logical
st	$reg_{rd}, [address]$	store word
stb	$reg_{rd}, [address]$	store byte
sub	$reg_{rs1}, reg\_or\_imm, reg_{rd}$	subtract
subcc	$reg_{rs1}, reg\_or\_imm, reg_{rd}$	subtract and modify icc

Table 3.1: Instruction description.

The terms used are:

- *reg_or_imm* is either reg_{rs2} or *simm13*.
- *reg* is an integer unit register. reg_{rs1} is the 5 bit rs1 field, which selects the first source operand. reg_{rs2} is the 5 bit rs2 field, which selects the second operand

and reg_{rd} is the 5 bit rd field, which selects a register to be the source for store instructions and the destination for all other instructions.

- *icc* is integer condition code.

- *address* is one of the following: reg_{rs1}, $reg_{rs1} + reg_{rs2}$, $reg_{rs1} + simm13$, $reg_{rs1} - simm13$, $simm13$ or $simm13 + reg_{rs1}$.

Register Windows

An important SPARC concept is *register windowing.* At any given time, a running program has access to 32 32-bit processor registers. These includes eight global registers, $g0$ to $g7$, and 24 further registers belonging to the active register window. Of these 24 registers, eight are local, eight are passed in from the previous window, and eight will be passed back from the next window as illustrated in Figure 3-2.

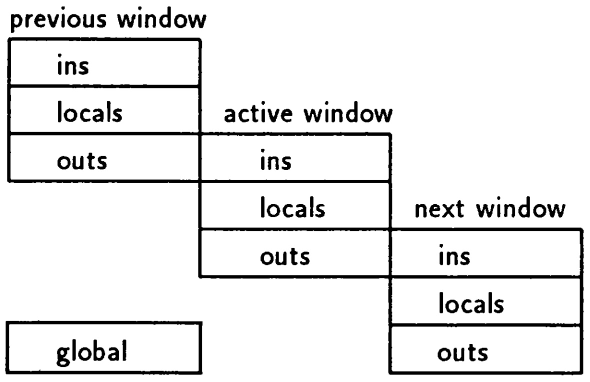

Figure 3-2: Register windows for the SPARC architecture. The diagram shows three different register windows with the overlapping parts for in and out registers.

One register window is reserved for traps or interrupts. A single instruction is used to switch in a new register window.

Up to six parameters can be passed as the other two registers are used to hold the return address and the old frame pointer. Additional parameters are passed on the stack.

When all register windows have been used, a trap occurs and one window is copied on the stack.

3.3 Performance

Architecture

The current SPARC architecture is divided into two families: The SS1 family (SS1, SLC, SS1+ and IPC) use the Fujitsu processor; the SS2 use the Cypress processor [Sun 1990]. SS1 and SLC have a 20MHz clock, SS1+ and IPC have a 25MHz clock and SS2 has a 40MHz clock. The Fujitsu processor provides seven register windows, the Cypress eight. All machines have a 64kbyte write-through cache [2] [Cockcroft 1991] with 16 byte cache line [3]. The drawback of write-through caches is the increased traffic between cache and main memory which slows down the system performance. The advantage is that cache and main memory are always consistent.

Instruction Timings

Most SPARC instructions can be executed in only one cycle if the assumption is true that instruction fetches, load instructions, and store instructions never have to wait upon the memory system. The instructions which cannot be executed in one cycle are single word load *ld* (two cycles), double word loads *ldd* and single word stores *st* (three cycles), and double word stores *std* and atomic load store instructions *ldstub* and *swap* (four cycles).

In most cases branches (which includes floating point branches) take one cycle, but on the Fujitsu chip untaken branches consume two cycles. For either chip if the delay slot of the branch was annulled an additional one cycle penalty will be incured since the delay instruction is still fetched. If the instruction following a delayed control transfer instruction cannot be filled with a useful instruction then an additional one cycle penalty is effectively incured.

[2]The term *write-through* is used to mean that the information written into the cache is also written to memory.

[3]The term *cache line* means the minimum number of bytes which will transfer between itself and main memory. For example, a four bytes memory request results in a 16 bytes cache read.

Jumps, indirect calls, returns, and trap returns take two cycles whereas direct calls take only one cycle. A trap incurs an additional three cycle penalty as the pipeline is drained and refilled.

Floating point instructions on the Weitek 3170 co-processor used in SS1, 1+, SLC, and IPC machines can use up to 118 cycles.

Memory Times

The timings given in subsection 3.3 all assume that instructions do not have to wait for the memory or bus system. Measuring memory access times is very difficult due to a number of factors such as cache size, cache organization and DRAM access times as well as interference by instruction fetches. The following timings are given for the SS1 family which has been used throughout this dissertation.

Assuming a cache hit no penalty is incured. Thus, for example, a single word load still takes two cycles. If a cache miss occurs the processor will stall while the address is translated and the memory is accessed. The stall lasts for twelve cycles while the line is loaded into the cache. For a store the stall lasts three cycles, causing a single store to take six cycles.

On the ELC and SS2, the cache miss costs are 26 cycles. This is mainly due to the SBus [4] used on this machines which runs at half the CPU clock rate.

3.4 Summary

Many of the features of the original RISC architecture have been implemented in the SPARC architecture. Most of the instructions can be executed in one cycle but instruction timings are radically effected by the need to use the memory bus. Processor stall times are about 12 cycles for the SS1 family if a cache miss occurs and increase with higher performance chips.

[4] *SBus* is the Sun bus system used on Sun machines.

Chapter 4

Compression Techniques

This chapter considers some techniques which may be suitable when compression on instruction sets is used to decrease object code size.

The aim of object code compression is to reduce the time to transmit the code over a channel of given bandwidth, or to remove redundant information. For the purpose of instruction encoding and decoding, only reversible, or lossless, compression is suitable as encoded instructions have to be recovered exactly from the compressed version. Several lossless compression techniques are in use to reduce the number of bytes needed to store or transfer information. Not all of them are suitable for encoding and decoding instruction streams as the encoding has to be done on static code, and decoding on dynamic code. Furthermore, for the purpose of encoding instruction streams only fixed models are suitable, as one cannot adapt the model during decoding in the same manner as during encoding. Techniques such as Ziv and Lempel's (LZ) algorithm cannot be used, even if it is one of the fastest decoding algorithm around, since LZ works with an adaptive dictionary.

There are two general methods for compression: *statistical* and *dictionary* coding. The better statistical methods are based on arithmetic coding, the better dictionary methods use LZ variants.

Furthermore, the compression process can be split into two separate parts: *Modelling* and *coding.* Models provide (probability) information about the data to compress, coding actually produces the compressed bit-stream. The model can split further into *non-adaptive*, or *fixed*, and *adaptive* models. Fixed modelling uses the same model for all data. Adaptive modelling, on the other hand, updates its model during encoding and decoding. *Semiadaptive* modelling uses a different model for each item of data being encoded.

Algorithms using a fixed set of compression information are more likely to reduce data size only for specific groups of data only such as numbers. In contrast, adaptive compression algorithms perform much better on a wide variety of data.

4.1 Non Suitable Compression Techniques

Several problems are encountered when using general compression methods [Held 1987] such as *null compression*, *bit mapping*, *run length compression*, *pattern substitution*, and *relative encoding* when integrated into computer systems:

1. Poor run time execution speed interferes with the attainment of very high data rates.

2. Most compression techniques are not flexible enough to process different types of redundancy.

3. Blocks of compressed data that have unpredictable lengths present storage space management problems.

Each compression strategy poses a different set of problems and consequently the use of each strategy is restricted in application. None of these schemes is an appropriate technique that can be used for efficient and effective object code compression as object code structure varies.

4.2 Potentially Suitable Techniques

4.2.1 Huffman Coding

One common element of the previously discussed data compression techniques is that they all operate upon symbol codes of a fixed bit size. Shortly after Shannon's work, Huffman discovered a way of constructing codes from a set of symbol probabilities which gives greater compression. Huffman coding [Huffman 1952] takes advantage of the probabilities of occurrence of single symbols and groups of symbols. Short codes can be used to represent very frequently occurring symbols while longer codes are used to represent less frequently encountered symbols. The statistical encoding process can be used to minimize the average code length of the encoded data.

The Huffman code is a "minimum redundancy" code that produces the shortest possible average code length given the symbols probability distribution and individual encoding of symbols. It can be shown that redundancy of Huffman codes, defined as the average code length less the entropy, is bounded by $p + 0.086$, where p is the probability of the most likely symbol [Bell *et al.* 1989]. Huffman codes also have a prefix property which means that no short code group is duplicated as the beginning of a longer group. An important property of the Huffman code is that it can be decoded instantaneously as the coded bits in the compressed data stream are encountered.

Huffman coding translates fixed-size pieces of input data into variable-length symbols. The standard Huffman encoding procedure prescribes a way to assign codes to input symbols such that each code length in bits is approximately $log_2(p)$ where symbol probability p is the relative frequency of occurrence of a given symbol expressed as probability. For example, if the letter Z is found to occur with a frequency of 0.1%, i.e $p = 2^{-10}$, of the time, it is represented by 10 bits.

In normal use, the size of the input symbol is limited by the size of the translation table needed for compression. A table is needed that lists each input symbol and its corresponding code. A second problem is the complexity of the decompression process. The length of each code to be interpreted for decompression is not known until possibly all but one bits are interpreted.

Huffman coding can be used to encode and decode instruction streams as encoding can be done on static code and decoding on dynamic code. The disadvantage is that Huffman coding can encode symbols to a minimum of one bit only. Often far less encoding is necessary and therefore Huffman coding has not been used for instruction set encoding.

4.2.2 A Universal Algorithm for Sequential Data Compression

One of the best known dictionary methods in compression is the algorithm developed by Ziv and Lempel [Ziv & Lempel 1977] and the variation [Ziv & Lempel 1978]. The algorithm developed is a universal algorithm for sequential data compression. In their scheme, pointers denote phrases in a fixed-sized window that precedes the coding position. There is a maximum length for substrings that may be replaced by a pointer, given by a parameter F (typically 10-20). These restrictions allow LZ to be implemented using a 'sliding window' of N characters. Of these the first $N - F$ have already been encoded and the last F constitute a *lookahead buffer.*

To encode a character, the first $N - F$ characters of the window are searched to find the longest match with the lookahead buffer. The match may overlap with the buffer but cannot be the buffer itself.

The longest match is then coded into a triple $< i, j, a >$, where i is the offset of the longest match from the lookahead buffer, j is the length of the match, and a is the first character that did not match the substring in the window. The window is then shifted right $j + 1$ characters, ready for another coding step. Attaching the explicit character to each pointer ensures that coding can proceed even if no match is found for the first character of the lookahead buffer.

Decoding is very simple and fast. The decoder maintains a window in the same way as the encoder, but instead of searching it for a match it copies the match from the window using the triple given by the encoder.

Advantages of the LZ compression technique are very fast decoding and good compression results on text and program source code. The disadvantage for the purpose of instruction set encoding is the adaptive property of the algorithm and the factor of only 1.5 which can be achieved for object code compression [Welch 1984].

4.2.3 Prediction by Partial Matching

The *Prediction by Partial Matching* (PPM) data compression algorithm developed by Cleary and Witten [Cleary & Witten 1984b] is capable of very high compression rates. Moffat [Moffat 1990] describes an implementation of the PPM scheme. In particular, he describes a variant that encodes and decodes at over four kbyte per second on a small workstation, and operates within a few hundred kilobyte of data space. The sheer magnitude of sample space for predictions using a long context makes them almost impossible to manage for practical compression. Even restricting the context to four prior characters will mean (using typically eight bit bytes) that there are in excess of four billion contexts possible. Moffat describes an alternative scheme. He makes the scheme adaptive, so the statistics will be built up as the stream of symbols is processed, without the need for a large model to be stored. This reduces the space requirements significantly.

Because short adaptive models are quick to establish useful statistics, but attain only limited compression, Moffat uses a model based on variable length context. At each coding step the longest previously encountered context is used to predict the next character. If the symbol is novel to the context, an escape mode is used and the context shortened by dropping one symbol. This process continues until the symbol is successfully transmitted.

The advantage of the PPM technique is that very good compression can be achieved. The disadvantage for the purpose of instruction set compression is the adaptive property of the algorithm.

4.2.4 Arithmetic Coding

Entropies associated with instruction sets are often much less than one bit per symbol. For their encoding and decoding a technique is needed that can get close to the entropy and is able to encode a symbol in less than one bit. Arithmetic coding (described in Chapter 5) is the technique which comes closest to meeting these requirements. Furthermore, arithmetic coding is suitable for different kinds of models, and therefore the compression scheme can be implemented using different order models if necessary.

The state of the art method of data compression is arithmetic coding [Langdon 1984, Witten *et al.* 1987], not the better known Huffman [Huffman 1952] method. It gives greater compression [Bell *et al.* 1990], and it clearly separates the model from the coding process. Arithmetic coding is a lossless coding technique that gives very high compression efficiency for a variety of data types, as well as being amenable to both software and hardware implementations [Bassiouni *et al.* 1988]. It actually approaches the theoretical entropy bound [Schoepke 1992f].

Chapter 5

The Implementation of Arithmetic Coding

Instead of replacing an input symbol with a specific code, arithmetic coding takes a stream of input symbols and replaces it with a single output number [Langdon 1984]. The encoding and decoding algorithms perform *arithmetic operations* on the code string. Symbols with high probability reduce the given range less than symbols with low probability. Probabilities close to one can be encoded very efficiently.

5.1 The Idea of Arithmetic Coding

In arithmetic coding a message is represented by an interval of real numbers on the number line between 0 and 1. As the message becomes longer, the interval needed to represent it becomes smaller, and the number of bits needed to specify that interval grows.

Successive symbols of the message reduce the size of the interval in accordance with the symbol probabilities generated by the model. Characters with high probabilities of occurrence (and, hence, larger intervals) have less effect on narrowing the interval than characters with small probabilities. Assigning larger intervals to the most frequent characters therefore increases the compression efficiency since more characters (on average) can be encoded in the same fixed–length field.

The commoner symbols reduce the range by less than rare symbols and hence add fewer bits to the message. The coding algorithm is symbolwise recursive; it operates upon and encodes (decodes) one data symbol per iteration or recursion. On each recursion, the algorithm successively constructs a subinterval of the real interval $[0, 1)$, and scales the subinterval to give it unit length, and identifies the new interval with $[0, 1)$. The location and length of the subinterval selected for this treatment is determined by the data symbol being encoded. By keeping track of this process, one obtains a representation of a string of symbols as a single subinterval of $[0, 1)$. Intervals may be stored via their end-points, or by specifying one end point and the length of the interval.

This procedure is invertible, essentially because the scaling and translation maps involved in the algorithm are all injective. Thus decoding is possible.

Before anything is transmitted, the range for the message is the interval between 0 and 1. As each symbol is processed, the range is narrowed to that portion of it allocated to the symbol. After seeing the first symbol the encoder narrows the interval to the range given by the probability of the symbol. The second symbol will narrow this new range down taking into account its allocation.

The most important properties of arithmetic coding are:

- It is able to code a symbol s_i with probability $p(s_i)$ in a number of bits arbitrarily close to $-\log_2 p(s_i)$.
- The symbol probabilities $p(s_i)$ may be different at each step, i.e. the encoding may be adaptive.
- It requires very little memory considering a zero–order model.
- There are very fast implementations.

5.2 Models for Arithmetic Coding

Arithmetic coding is suitable for both fixed and adaptive models. The simplest kind of model is one in which symbol frequencies are fixed (*fixed* model). This model can be calculated and transmitted before the message is sent. Performance of the fixed model can be improved by ordering the ordinary fixed model in the cumulative frequency table according to the symbol frequency. The so called *sorted* model achieves much better performance. In both cases (fixed and sorted model) an exact model is used which was built by a previous run over the source to be coded. This minimizes the effect of additional bit shifting operations during the encoding and decoding process, which could influence the performance [Witten *et al.* 1987].

An *adaptive* model [Abrahamson 1989] uses the changing symbol frequencies seen so far in a message. The model itself is updated as each symbol is seen. Provided both encoder and decoder use the same updating algorithm, their models will remain in step. The encoder receives the next symbol, encodes it, and updates the model. The decoder identifies it according to its current model and then updates its model. It is shown by Cleary and Witten [Cleary & Witten 1984a] that, under quite general conditions, the fixed model will not give better overall compression than the adaptive model. As the adaptive model cannot be used for encoding and decoding instruction streams, no further information is given.

5.3 Simple Character Set Example

The following example demonstrates the main idea of arithmetic coding and how it works. Consider a simplified set of three characters whose probabilities of occurrence and their assigned intervals are given in Table 5.1. It is assumed that the character "@" will be used only to indicate the end of an encoded portion of data and therefore cannot occur in the input data itself.

Simplified Character Set

character	*probability*	*interval*
A	0.4	[0.0, 0.4)
1	0.4	[0.4, 0.8)
@	0.2	[0.8, 1.0)

Table 5.1: Simplified character set with probability and assigned interval.

The intervals in arithmetic coding are nonoverlapping as seen in Table 5.1. Starting the encoding process with the initial interval $[0.0, 1.0)$ the interval then narrowed repeatedly (as characters are processed), thus each interval is totally contained in the preceding one. Supposing that one encodes the two–character message "1A": Upon receiving the first character, the initial interval is transformed into the new interval $I_1 = [0.4, 0.8)$ which corresponds to the code range of character "1" in Table 5.1. During encoding of the second character the interval I_1 is narrowed to $I_2 = [0.4, 0.56)$ which represents a subinterval of I_1 corresponding to the range of character "A".

Generally, if $I_k = [a_k, b_k)$ is the current interval, and the range of the next character is given by $[s, f)$, the next interval $I_{k+1} = [a_{k+1}, b_{k+1})$ is computed as follows:

$$a_{k+1} = a_k + s(b_k - a_k) \tag{5.1}$$

$$b_{k+1} = a_k + f(b_k - a_k) \tag{5.2}$$

Thus I_{k+1} is a subinterval of the interval I_k. Furthermore, the following relationship holds true:

$$length(I_{k+1}) = (f - s) length(I_k) \tag{5.3}$$

where

$$length(I_k) = b_k - a_k. \tag{5.4}$$

The encoding string "1A" is transformed from the interval

$$I_2 = [0.4, 0.56) \tag{5.5}$$

to the final interval

$$I_3 = [0.528, 0.56) \tag{5.6}$$

by appending the end of message character "@".

The decoding process of the above message uses primarily the same logic. When the decoder receives the value 0.53, it knows that the first character has to be "1" since the value 0.53 is contained in the interval $[0.4, 0.8)$ corresponding to the character in

Table 5.1. By examining the next value the decoder concludes that the next character has to be "A" since this is the only character which can transform the interval into an interval containing the value seen. The decoder continues in this fashion until it finds the end of message character.

The decoder does not really need to know both ends of the range (or one end and the range). Instead, a single number within the range is suffice as the decoding ends with the end of message character.

During encoding, the code range narrows and the top bits of the two values representing the interval become the same. Since these bits cannot be affected by future narrowing, they can be sent immediately and new bits can be shifted into the encoding unit. However, precautions have to be taken to prevent a precision, or underflow, problem.

5.4 The Conventional Algorithm

The algorithm described by Witten et.al. [Witten *et al.* 1987] is based on integer rather than floating point arithmetic. The intention was to simplify necessary arithmetic operations and to speed up the coding process.

The encoding process is described as *incremental transmission.* The top bits of the lower and upper bound can be transmitted immediately if they are the same, since future narrowing cannot affect them. As a result a stream of bits flows out of the encoder.

During the decoding process (*incremental reception*) a stream of bits is fed into the decoder. After identifying the symbol the decoder shifts out equal bits from the upper and lower bound. New bits from the encoded bit stream replace them.

During decoding a search loop is used to find the symbol concerned. The search is insufficiently fast for more distributed data (such as object code) even if the symbols are ordered according to their probability so that less time is needed to find frequent symbols and more time to find rare symbols. A faster search algorithm is necessary to decode object code.

5.5 A Fast Decoding Algorithm for Arithmetic Coding

Here I present a fast decoding algorithm for arithmetic coding developed in co-operation with Geoff C Smith which utilizes a fixed model. Decoding performance improves by up to 18%, depending on the target data, with an average of 7.47%. The algorithm's performance is never worse than that of the rival techniques when applied to input data from the Calgary corpus described in Table 5.2 [Bell *et al.* 1990]. The improvement has been achieved by using fast look-up tables (of size only four kbyte) instead of the original search algorithms used in [Witten *et al.* 1987].

Calgary Text Compression Corpus

abbrev	source
bib	Bibliographic files (refer format)
book1	Hardy: Far from the madding crowd
book2	Witten: Principles of computer speech
geo	Geophysical data
news	News batch file
obj1	Compiled code for VAX: compilation of progp
obj2	Compiled code for Apple Macintosh: Knowledge support system
paper1	Witten, Neal and Cleary: Arithmetic coding for data compression
paper2	Witten: Computer (in)security
pic	Picture number 5 from the CCITT Facsimile test files (text+drawings)
progc	C source code: compress version 4.0
progl	Lisp source code: system software
progp	Pascal source code: prediction by partial matching evaluation program
trans	Transcript of a session on a terminal

Table 5.2: A brief description of the Calgary text compression corpus as used throughout this thesis is given.

The Calgary corpus is used in the book *Text Compression* [Bell *et al.* 1990] and several other researchers are now using it to evaluate text compression schemes. The corpus represents nine different types of text, and to confirm that the performance of schemes is consistent for any given type, many of the types have more than one representative. Details of the individual texts are given in [Bell *et al.* 1990], a brief summary is given in Table 5.2.

The decoding process of arithmetic coding is about 30% slower than encoding

[Witten *et al.* 1987], due to the difficulty of finding symbols in the cumulative frequency table. For the adaptive model Witten et.al. [Witten *et al.* 1987] have shown that during decoding between 33% and 39% (depending on the data) of the time is spent calculating cumulative frequency and finding the correct symbol via a search loop.

Table 5.3 shows the number of search loops necessary for nine out of 14 different source codes from the Calgary corpus using the fixed sorted model. It is worth noting that the worst problem is in decoding compressed object code, the particular area of interest.

Search Loop Executions

program	*average number*
bib	14
book1	8
geo	36
news	14
obj1	37
paper1	12
pic	2
progc	14
trans	17
average	17

Table 5.3: Average number of search loop executions during decoding to find the correct symbol in the cumulative frequency table.

The data given in Table 5.3 and in particular the poor decoding performance on object code were the main reasons for constructing an improved decoding algorithm [Schoepke & Smith 1993].

A modification of the algorithm was sought which was significantly faster in decoding, and did not incur an unacceptable space overhead.

5.5.1 The Mathematics Behind Look-Up Tables

In the model used there are 257 subintervals which partition the unit interval, and the elements of this partition will typically have differing lengths, and represent different symbols. The 257 symbols involved are the ASCII characters together with an "end of message" symbol. The dynamic representation of the encoded message is a number in one of the subintervals of the partition. How does one tell which subinterval in the partition includes the message subinterval?

It clearly suffices to identify the subinterval in the partition which includes any given point of the message subinterval. This can be done by a divide and conquer search taking $log_2 256 = 8$ steps.

The explanation has been in terms of real intervals, but in practice one must render this process discrete. The rôle of the unit interval is replaced by the integers in the range $[0, 2^{14})$. This particular number range is selected to be consistent with Witten's implementation [Witten *et al.* 1987], and thus facilitate a realistic comparison of the algorithms. The encoded message is at all times actually a single number rather than an interval, and at each stage of the decoding process the encoded message number is recalculated by a process analogous to scaling and translation.

This range is divided into 2^{11} ranges of length 2^3. All bar at most 256 of these ranges correspond to a well-defined symbol to be decoded. At most 256 of these ranges leave the issue unresolved, the problem being that the range of length 8 is straddling at least one boundary.

Thus one makes a primary look-up table of 2^{11} rows, each one pointing either to the symbol to be decoded, or if this is not resolved, to a secondary *extension* table giving a precise description of the neighbourhood in question. Each secondary extension table will have 2^3 rows.

Thus, to find the next symbol to be decoded, the encoded message string is taken, which might be 11100011100011 and look at the first 11 binary digits of this 14 digit binary. This is 11100011100. The primary look-up table is used to find the symbol to be decoded. In the event of failure, an extension table with 2^3 entries has to be used. One has to look at 011 in this table to determine the symbol to be decoded.

The sizes of the primary table and the extension tables have been selected to minimize the overall space overhead. One justifies this assertion under the assumption that all 256 extension tables will actually be needed.

The range has length 2^{14}. One divides the range into 2^{14-n} intervals of length 2^n.

Thus one needs a primary look-up table of length 2^{14-n} and at most 2^8 extension look-up tables of total length 2^{n+8}. Thus the total length of all the look-up tables is $2^{14-n}+2^{n+8}$. This is minimized when $n = 3$, when one needs a total table size of 2^{12} rows.

In fact experiment indicates that these values for the sizes of the tables are reasonably robust, and that the simplifying assumption is not leading far astray. Suppose that, in practice, only 64 secondary tables are needed. In this case, one should seek to minimize $2^{14-n} + 2^{n+6}$ which happens when n is 4. One now needs a total of only 2^{11} rows. The saving in space is 50%. Since the space in question is rather small, it is unclear that such an optimization is worthwhile.

One could, of course, refine this technique still further, and have deep nests of look-up tables. For this purposes the saving in space is not really justified, since the necessary extra table-chasing will degrade the speed of the decoding process.

5.5.2 Implementation

Our implementation for look-up tables uses a primary table of size 2^{11}, and 2^8 extension tables, each of size 2^3. The tables were built automatically by examination of the exact model. The time needed for the construction of the tables is included in the results, even though this is a one-off penalty. From the performance point of view it would be nice to get the right symbol out of the look-up table with one operation.

Of course, one could build a single comprehensive look-up table of size 2^{14}, but this is an unacceptable space overhead. In the look-up table there are 256 "bad points" which have to be considered. If it is not possible to track down the correct symbol via the primary look-up table (i.e. this is a "bad point"), one has to look at an extension table. This second operation slows down the whole process by about 2% as the information one is looking for has to be addressed for a second time. Table 5.4 gives the information about the percentage of *misses* for different kinds of input data. A *hit* means one can use the entry given in the primary look-up table. A *miss* means one has to look in the look-up extension which is more expensive since one must examine a secondary table.

Even for the particular interest, object code, the secondary table is only used in 15% of all cases.

Miss Distribution	
program	*percent*
bib	6.65%
book1	3.58%
geo	15.64%
news	5.74%
obj1	15.63%
paper1	5.98%
pic	6.00%
progc	7.11%
trans	4.94%
average	7.92%

Table 5.4: Miss distribution for different input data given in percentages.

5.5.3 Experimental Results

The experimental results shown in Table 5.5 are based on input data from the Calgary corpus and measured on a Sun SPARC ELC (25MHz), running SunOS 4.1.1 Rev B with the gcc compiler [Stallman 1989] version 1.40 using optimizer option -O. The interference of I/O is minimized by generating no output data during encoding and decoding. The timings shown are collected by using the UNIX `time` command, the program size by the UNIX `ls` command.

The experimental results are based on the C implementation described by Witten et.al. as mentioned before [Witten *et al.* 1987]. The time needed to read the model is included in the results, but is negligible. The time needed to sort the model is again included in the results, but they did not have any visible effect (after allowing for experimental errors). The encoding times collected (not printed) confirm this.

The new algorithm performs best on object code and data which is difficult to compress such as that in **geo**. This is not surprising as the number of loop executions needed to find the symbol during decoding increases with more random data (see Table 5.3). In fact, using data which is random one can improve performance by more than 50%. Table 5.5 shows the original file size in bytes, the remaining percentage after encoding, timing for the conventional and improved algorithm, and the improvement achieved

Performance Improvement

name	size	encoded	conventional algorithm	improved algorithm	improvement achieved
	bytes		*μs/byte*	*μs/byte*	*percent*
bib	111 261	65.0%	35.95	33.26	7.48%
book1	768 771	56.6%	32.91	31.22	5.14%
book2	610 856	59.9%	33.72	31.43	6.79%
geo	102 400	70.6%	41.99	34.18	18.60%
news	377 109	64.9%	35.80	32.88	8.16%
obj1	21 504	74.4%	41.85	37.20	11.11%
obj2	246 184	78.5%	44.16	36.06	18.34%
paper1	53 161	63.3%	33.86	33.86	0%
paper2	82 199	57.5%	32.85	31.63	3.71%
pic	513 216	15.2%	22.04	22.04	0%
progc	39 611	65.0%	35.34	32.82	7.13%
progl	71 646	59.6%	33.50	32.10	4.18%
progp	49 379	60.9%	34.43	32.40	5.90%
trans	93 695	69.2%	37.36	34.15	8.59%
average	224 356	61.47%	35.41	32.52	7.47%

Table 5.5: Comparison of decode times using the conventional and improved decoding algorithm. The program size, the remaining percentage after encoding, the decoding times of uncompressed data, and the improvement achieved is given. The times exclude I/O. The figures give *user time* from the UNIX `time` command and program size from the UNIX `ls` command.

with the new algorithm.

Allowing for experimental errors, the encoding times (not printed) are equal for all three fixed models as there is no change in the encoding algorithm (except in the sorted case, though the time impact of this variation is nugatory).

5.5.4 Summary

With the algorithm presented it is possible to improve decoding performance for arithmetic coding utilizing fixed models by up to 18% measured against a sorted fixed model. The results presented show clearly the improvement gained with only four kbyte of additional memory. Especially for data which is difficult to compress, notably object code, the algorithm performs extremely well compared with the rival technique. This favours the new algorithm for this purpose.

Furthermore, with the improved algorithm, hardware implementation is much easier as no search algorithm has to be implemented. Instead, finding a symbol in the cumulative frequency table requires one comparison only with the one-off penalty of four kbytes of additional memory.

5.6 Hardware Implementation

Due to the complexity of most compression methods, past implementations of data compression techniques have mostly been restricted to software. Only a few hardware designs using associative memory [Lea 1978], and microprocessor based systems [Hawthorn 1982] have been reported. A proposed design for compression by textual substitution is given in [Gonzalez-Smith & Storer 1985] and a brief discussion of the hardware design of the LZW algorithm using hash tables is given in [Welch 1984]. A fast VLSI implementation of the Huffman's scheme is given in [Mukherjee & Bassiouni 1987] and a high level description of a VLSI chip for the implementation of a modified arithmetic coding scheme is given in [Bassiouni *et al.* 1988]. A hardware implementation of the Q-Coder is described by Mitchell and Pennebaker [Mitchell & Pennebaker 1988].

The arithmetic coding scheme consists of arithmetic operations such as addition and multiplication and hence can be implemented in hardware. Parallel and very wide buses can be used on-chip without regard to pinout limitations. Clock rates can reach a few gigahertz within a chip module, because the dimensions are very small and reflections die out quickly [Storer & Szymanki 1982].

5.7 Summary

In arithmetic coding a suitable coding technique for encoding and decoding an instruction stream efficiently and effectively has been found. With this technique encoding as close to entropy as desired is possible. The technique can be implemented in hardware which is a necessary feature for the purpose of this work. Furthermore, the fast decoding algorithm developed gives a significant speed advantage over the rival technique, especially for object code as shown in Table 5.5.

Chapter 6

Experimental Results

Experimental results gathered on the SPARC architecture are essential to deal with instruction dependency and to exploit the structure given in the instruction stream [Glass 1991, Sun Microsystems Inc. 1987]. The experiments were made regarding the information necessary to encode and decode an instruction stream efficiently and effectively. Information about SPARC field distributions and their entropy was gathered for this purpose.

6.1 Experiments Performed

The workload for the experiments performed is built with a set of real programs that includes a C compiler (GNU gcc [Stallman 1989]), a statistics program, a word processing system (TEX), part of the X window system, the command interpreter (tcsh), a terminal emulation program (xterm), and a debugger (GNU gdb). These programs have been chosen not only because of their size but also because of their availability. The quantity of information that needs to be gathered about the dynamic characteristic of program execution depends very much on the cost one is willing to pay for the information. Sample programs from which the dynamic characteristics of programs are collected are a C compiler (GNU gcc [Stallman 1989]), a statistics program, the command interpreter (tcsh) and a debugger (GNU gdb), because the results could be collected relatively easily. All these statistics are program and compiler dependent, as different classes of applications typically use different language features [Weiker 1984]: Numeric programs frequently use floating-point arithmetic and often operate on arrays; business application programs are mostly dominated by I/O activities; system program often use enumeration, record, and pointer data types. Moreover, it is unavoidable that such a study will also measure the ability of a compiler to generate efficient code.

Statistics about static characteristics of programs are based on more than 500000 instructions. The statistics of the dynamic characteristics are based on more than ten million instructions executed. The code is compiled for SPARC. All the presented data are average values from the above programs. Special program behaviour is pointed out separately. Floating-point and co-processor operations are not the subject of this study.

Instead, the purpose of this study is to gather information about the usage of instruction set constructs. Information that has been found to be useful for this purpose includes

- Distribution of opcodes;
- Distribution of opcode pairs;
- Distribution of registers (*global*, *local*, *in* and *out* register);
- Distribution of constants;
- Distribution of branches.

Static information about instruction sets can be obtained by writing a special program analyzer or by modifying an existing compiler. This data was collected by a program analyzer counting the number of occurrence of each instruction achieved from the relevant instruction fields. The relevant addresses of executable program parts were taken from the symbol table.

Dynamic information about instruction sets can be obtained by either writing an interpreter or by writing a trace program. Of course, tracing a program is very expensive; the time to trace a program and analyze the data produced is several orders of magnitude greater than the time required to simply execute the program to be traced. The data presented here was collected by tracing the program.

6.2 Empirical Analysis

6.2.1 Opcodes

Static Characteristic			Dynamic Characteristic		
opcode	*percent*	*cum.per.*	*opcode*	*percent*	*cum.per.*
sethi	19.71	19.71	ld	12.36	12.36
ld	15.54	35.14	sethi	12.21	24.57
or	14.07	49.22	add	11.49	36.06
call	5.72	54.94	subcc	8.33	44.39
st	5.26	60.20	or	8.30	52.69
subcc	4.54	64.74	bne	5.33	58.02
add	4.53	69.27	orcc	4.37	62.39
orcc	3.65	72.92	sll	3.91	66.30
ba	3.64	76.56	be	3.73	70.03
be	3.36	79.93	ldsb	3.47	73.50
bne	3.04	82.96	st	2.90	76.40
sll	1.95	84.91	ba	2.53	78.93
jmpl	1.52	86.43	stb	1.82	80.75
sub	1.48	87.91	and	1.59	82.34
ldsb	1.19	89.10	jmpl	1.58	83.92
restore	1.07	90.17	call	1.54	85.46
and	0.97	91.14	srl	1.52	86.98
stb	0.89	92.03	bl	1.47	88.45
ldub	0.74	92.77	andcc	1.31	89.76
lduh	0.73	93.50	sub	1.23	90.99

Table 6.1: Distribution of twenty most frequently used opcodes given in percentage and cumulative percentage for static as well as dynamic case.

The static as well as dynamic frequency of opcodes is provided in Table 6.1. Only a small number of opcodes occurs with high frequency [Bennett & Smith 1989]. The instruction *sethi 0, %g0* is equivalent to *nop* and *or %g0, register-or-immediate, register* is equivalent to *move register-or-immediate, register* where *%g0* is the global register zero. For the static case just three opcodes (*sethi*, *ld* and *or*) together account for nearly 50% of all opcodes. Just sixteen opcodes account for more than 90%.

In the dynamic case the observation is more spread. The opcodes *ld*, *sethi* and *add* account for more than 36% and twenty opcodes account for more than 90% of all opcodes executed. The detailed figures (not printed) show there is no great difference between distinct applications. These results match very well with earlier work [Alexander & Wortman 1975].

In static as well as dynamic cases with a small number of opcodes most of the compiled code can be built thus a reduced instruction set is a valid approach.

6.2.2 Opcode pairs

Static Characteristic				**Dynamic Characteristic**			
pred.	*succ.*	*percent*	*cum.per.*	*pred.*	*succ.*	*percent*	*cum.per.*
sethi	ld	7.18	7.18	sethi	ld	3.76	3.76
sethi	or	6.44	13.62	sethi	or	2.81	6.57
call	sethi	4.86	18.48	ld	add	2.29	8.86
sethi	sethi	3.29	21.77	add	st	2.07	10.93
or	call	3.28	25.05	orcc	bne	1.87	12.80

Table 6.2: Distribution of five popular opcode pairs given in percentages (*percent*) and cumulative percentages (*cum.per.*) of all pairs examined. *Pred.* and *succ.* describe predecessor and successor respectively.

The frequencies of different SPARC opcode pairs are given in Table 6.2. Just five pairs account for more than 25% of all pairs in the static case and about 13% in the dynamic case. Each of these high frequency pairs is a potential candidate for becoming a single opcode in an improved computer architecture [Sweet & Sandman 1982]. This result in particular shows the dependency between instructions.

Foster and Gonter [Foster & Gonter 1971] got equivalent results for their observations of successors, namely that an opcode is followed mainly by one of seven opcodes. The results provided by Alexander and Wortman [Alexander & Wortman 1975] are similar although they got a higher percentage in the dynamic case.

6.2.3 Registers

For a RISC architecture it is very important to get good support for registers because of its register structured architecture. The SPARC architecture provides seven banks of windowed registers: *global*, *local*, *in* and *out* registers and *floating-point* registers. The usage of floating point registers is application dependent and therefore not discussed here. A co-processor was not installed.

Static Characteristic

register	*no. 0*	*no. 1*	*no. 2*	*no. 3*	*no. 4*	*no. 5*	*no. 6*	*no. 7*	*sum.*
global	91.60	8.24	0.08	0.05	0.02	0.01	0.00	0.00	100.00
local	21.93	17.48	12.01	11.62	10.23	9.11	8.46	9.17	100.01
in	17.05	9.06	7.40	6.93	8.94	16.05	*30.29*	*4.27*	99.99
out	40.55	19.71	11.51	8.26	6.67	5.80	*2.87*	4.63	100.00

Table 6.3: Distribution of registers for the static case given in percentage for each register bank separately.

Dynamic Characteristic

register	*no. 0*	*no. 1*	*no. 2*	*no. 3*	*no. 4*	*no. 5*	*no. 6*	*no. 7*	*sum.*
global	84.35	15.04	0.62	0.03	0.00	0.00	0.00	0.00	100.04
local	5.92	8.37	25.18	8.14	13.18	6.34	22.39	11.50	101.02
in	11.54	12.13	8.15	10.39	12.26	15.14	*27.12*	*3.26*	99.99
out	33.41	18.72	11.63	8.72	9.06	13.30	*2.53*	3.87	101.24

Table 6.4: Distribution of registers for the dynamic case given in percentage for each register bank separately.

The examination of register usage (including registers as source and destination) shows some interesting results. Table 6.3 provides the information for the static case [1]. The eight global registers provided by the Sun SPARC architecture are seldom used in the static case with the one exception of global register zero which accounts for 91.6% of global register use. The explanation is that global register zero always contains zero and is therefore often used to set other registers to zero. A surprising observation is that global register seven was never used in all sample programs. Global register six is used three times and then only if register optimization is in use. Therefore it is not shown in Table 6.3.

It is a different case for *in*, *out*, and *local* registers which are used with higher frequencies. The first three *out* registers account for more than 70% of eight *out* registers. *In* register number six is the frame pointer and *in* register number seven is the return

[1]Cumulative percentage does not sum to 100.00% due to rounding and truncation of data.

address. Not surprisingly the frame pointer is the most used *in* register with more than 30% in the static case. The distribution of local registers in the static case is more or less as expected: Registers with lower numbers are more likely to be used than registers with higher numbers.

Table 6.4 provides the information for the dynamic case. *Global* registers four and five, which are rarely used in the static case, are not mentioned in the dynamic case, because they occur only in the X window system, which is not used for dynamic characteristics. The results of *global* registers are quite comparable to the static case. *In* and *out* registers are primarily used for passing parameters to and from subroutines and the results lead us to the assumption that usually only a few parameters are passed between procedures in the sample programs. Tanenbaum [Tanenbaum 1978] discovered that the average number of parameters is 1.5 (2.0) for the static (dynamic) case for SAL programs. The overwhelming majority (at least 97%) of procedures in system code take fewer than six parameters and the average number of parameters, measured statically and dynamically is no greater than 2.1 in any of the studies cited by Weicker [Weiker 1984].

The statistics represent only average values. The property of each sample program could be quite different: e.g. the statistics program uses local register two much more frequently than the average value provided in Table 6.4 suggests. The explanation is that a loop variable which is used extremely frequently is stored in this particular register. The same explanation applies for the program from the operating system (tcsh).

	Register			
characteristic	*global*	*local*	*in*	*out*
static	32.73%	11.51%	22.77%	32.99%
dynamic	30.52%	11.74%	11.47%	46.27%

Table 6.5: Distribution of all registers for static and dynamic case.

Table 6.5 shows the results for register usage as a whole. It is a good illustration as to which register bank is used most and it is shown that there is quite a difference between static and dynamic observations regarding *in* and *out* registers. Since the caller's *out* register become the callee's *in* register the results show that more parameters are passed to the callee's than results received from them.

6.2.4 Constants

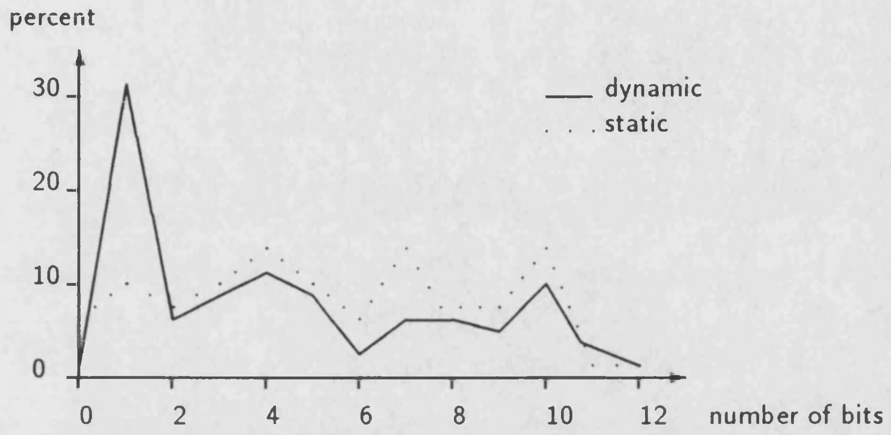

Figure 6-1: The distribution of positive immediate values is shown. The **x** axis shows the number of bits needed to represent the magnitude of an immediate value – 0 means the immediate field value was 0.

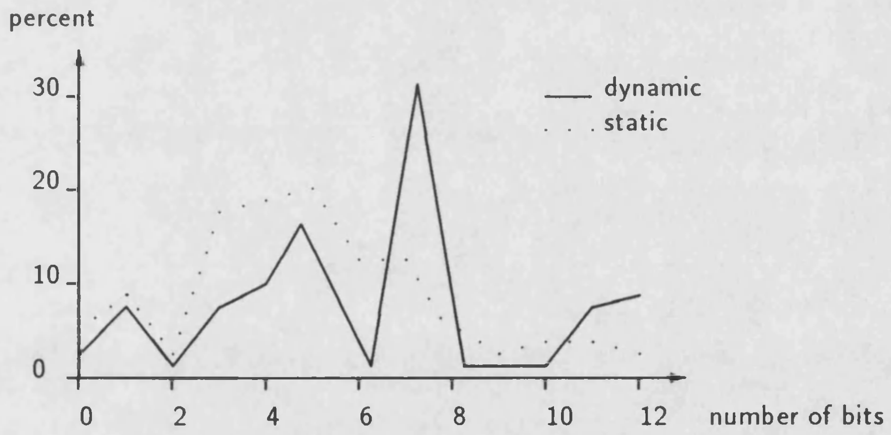

Figure 6-2: The distribution of negative immediate values is shown. The **x** axis shows the number of bits needed to represent the magnitude of an immediate value – 0 means the immediate field value was 0.

Those instructions dealing with constants use a signed immediate constant that fits in 13 bits. Figure 6-1 provides the results for positive constant arguments [2]. Different results than other program statistics are obtained [Alexander & Wortman 1975, Sweet & Sandman 1982] presumably because of the different way of setting registers to zero. The results provided in Table 6.3 and Table 6.4 reinforce this assumption. In any case, it is interesting to note that in the static case 46.12% of all positive constants could be represented using 4 bits and 75.59% using 7 bits or less.

In the dynamic case the results are even more significant. 48.80% of all positive constants could be represented using 3 bits or less. With five bits or less one can represent about 70% of all positive constants. The high frequency of the constant one in the dynamic case suggests that an instruction dealing with this particular constant could be useful.

Results for negative constants are provided in Figure 6-2 and they are similar. In the static as well as the dynamic case more than 80% of all constants are between $-[2^0 - 2^7)$.

6.2.5 Branch Instructions

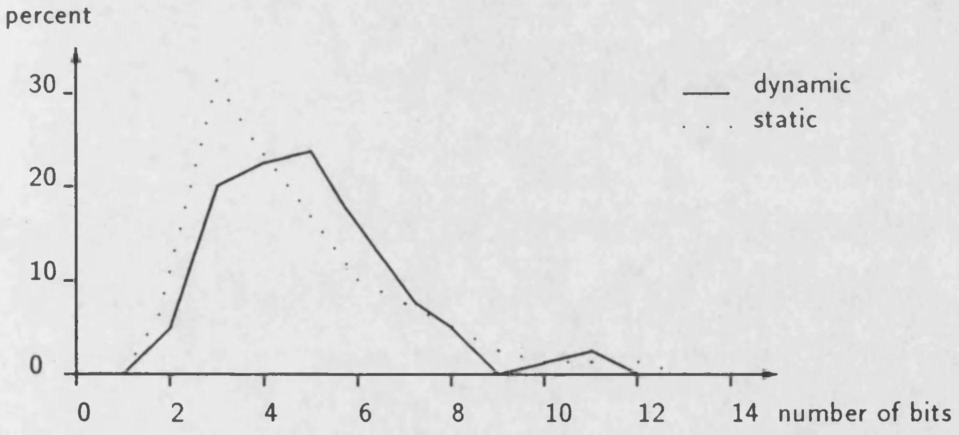

Figure 6-3: The distribution of positive displacement values is shown. The x axis shows the number of bits needed to represent the magnitude of a displacement value.

Branch instructions build their destination by multiplying the contents of the instruction field by four and than adding it to the PC (Program Counter) to calculate the destination. The branch instruction causes a PC-relative, delayed control transfer to

[2]The immediate field of the *sethi* instruction is not included here.

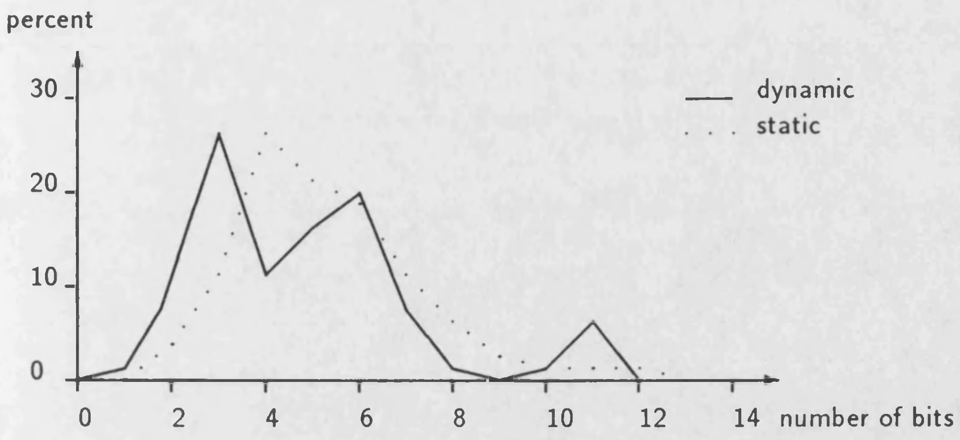

Figure 6-4: The distribution of negative displacement values is shown. The **x** axis shows the number of bits needed to represent the magnitude of a displacement value.

the address

$$PC + (4 * sign\_extend(distance)).$$

The sign_extend distance bitfield is 22 bits long. The distances which are given in Figure 6-3 and Figure 6-4 are the results collected by the bit field of all branch instructions.

The static analysis of positive branch instructions shows that about 60% of the branch destinations fit into four bits or less and about 85% of them fit into six bits or less. With only ten bits one can describe 99% of all positive branch destinations.

The observation in the dynamic case is similar. With four bits one can describe about 47%, with six bits about 85% and with ten bits about 98% of all positive branch destinations. There is no positive branch destination with more than 14 bits in the sample programs in the static or the dynamic case.

The distribution of negative branch destinations is similar. In the static (90.17%) as well as the dynamic (92.89%) case more than 90% of all branch destinations use a distance between $-[2^0 - 2^7)$.

Therefore most branch instructions use destinations that are near to the location of the branch instruction itself. This result depends very much on the compactness of the code emitted by the compiler. But it shows that there is great redundancy in branch fields because they comprise 22 bits for the SPARC architecture.

Alexander and Wortman's [Alexander & Wortman 1975] observations during their study of XPL programs are similar: they found that 54.4% of the branches were no more than 128 bytes away from the location of the branch instruction.

Branch Instructions

opcode	*operation*	**static**	**dynamic**
ba	branch always	3.64	2.53
bcc	branch on carry clear	0.16	0.39
bcs	branch on carry set	0.15	0.34
be	branch on equal	3.36	3.73
bg	branch on greater	0.21	0.20
bge	branch on greater or equal	0.53	0.56
bgu	branch on greater unsigned	0.17	0.92
bl	branch on less	0.52	1.47
ble	branch on less or equal	0.35	0.24
bleu	branch on less or equal unsigned	0.10	0.24
bn	branch never	0.00	0.00
bne	branch on not equal	3.04	5.33
bneg	branch on negative	0.02	0.35
bvc	branch on overflow clear	0.00	0.00
bvs	branch on overflow set	0.00	0.00

Table 6.6: Distribution of all branch instructions in alphabetic order given in percentage.

More detailed information about branch instruction is provided in Table 6.6. Together, *ba*, *be* and *bne* account for about 10% (12%) of all opcodes and about 82% (71%) of all branch instructions in the static (dynamic) case, respectively. All other branch instructions (except *bl* in dynamic case) are used not at all or only rarely.

6.3 Entropy

Figure 6-5 shows the entropy for both the static and dynamic instruction stream. Graph a) shows the instruction stream treated as eight bit symbols, graph b) looks at just the opcode of the instruction. For eight bit long symbols as well as opcodes entropy decreases significantly to about 0.2 bits for the fifth order Markov source from the zero order Markov source in the dynamic case [Schoepke 1992b]. These results reflect the considerable structure in compiled code and show there is great redundancy in such instruction streams and great dependency between instructions following each other which leads to very high conditional probabilities and thus to very low entropies. Such entropies as shown in Figure 6-5 imply one can generate more compact code [Schoepke 1992h] because the entropy $H(S)$ gives the number of bits for optimal encoding. The decrease is of about 90% in entropy for the fifth order Markov source from the zero order Markov source for symbols in the static case. In the dynamic case one achieves even lower entropy.

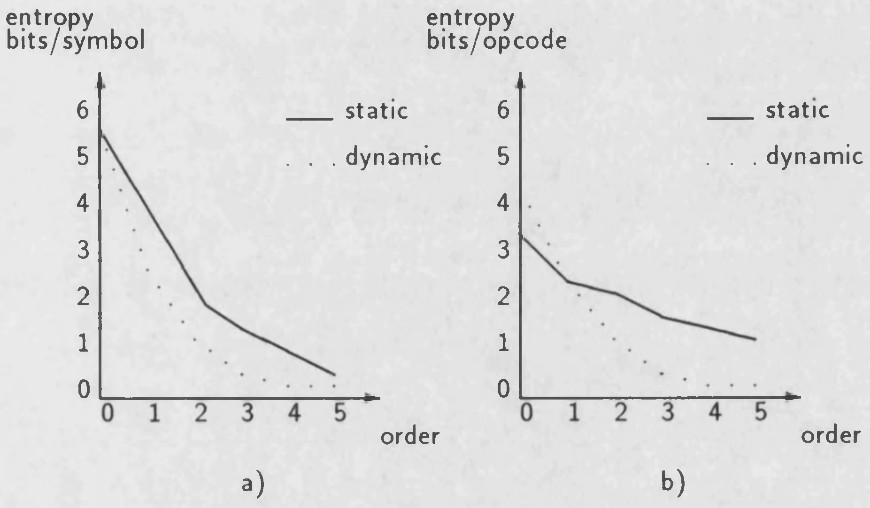

Figure 6-5: Average entropy of m^{tm} order Markov sources in bits per eight bit long symbol and bits per opcode for static as well as dynamic case.

Redundant information can also be found in addresses, and, furthermore, as address space and their corresponding address words have grown in size from 16 to 24 to 32 to 64 bits, so has the percentage of a given address word containing redundant information [Farrens & Park 1991].

One can exploit the structure of the instruction stream and calculate the entropy for different instruction fields.

The results shown in Figure 6-6, Figure 6-7, Figure 6-8, and Figure 6-9 confirm the view that instruction streams have considerable structure. The graphs shown provide information for register fields *rs1*, *rs2*, and *rd* as well as *cond* and *imm22* bit fields in both static and dynamic cases. In the dynamic case the entropy drops much more sharply than in static case. In the case of register fields (which includes registers as

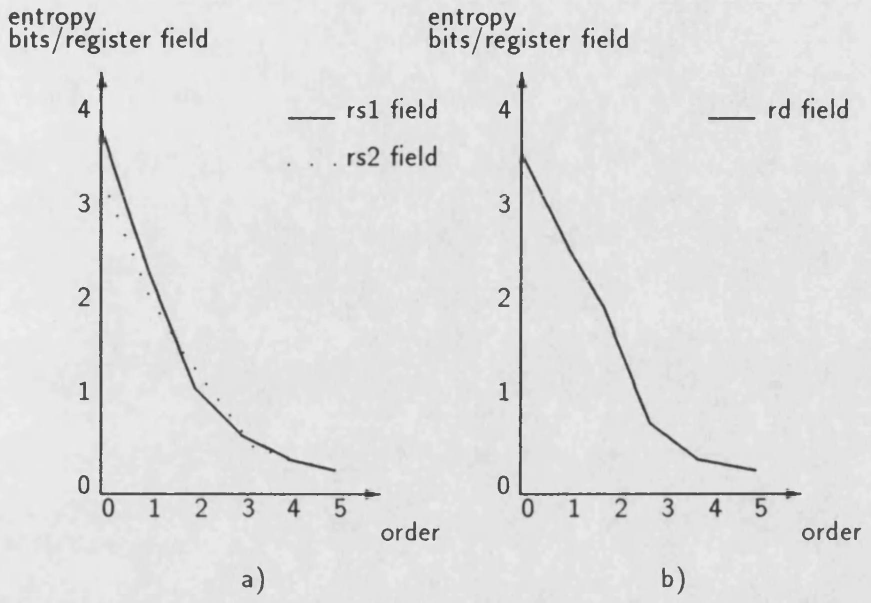

Figure 6-6: Average entropy of m-th order Markov source in bits per register field for the dynamic case. a) shows the register fields $rs1$ and $rs2$, b) shows the register field rd.

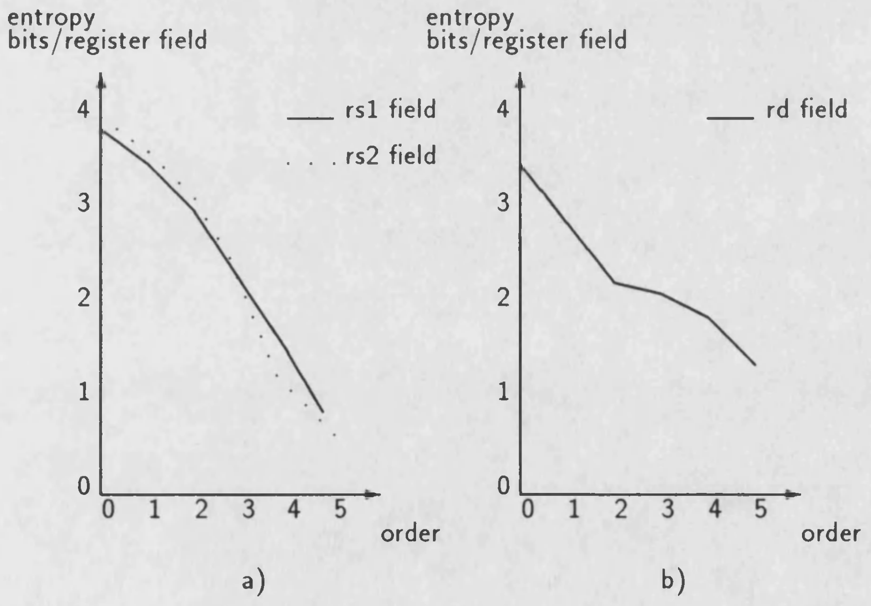

Figure 6-7: Average entropy of m-th order Markov source in bits per register field for the static case. a) shows the register fields $rs1$ and $rs2$, b) shows the register field rd.

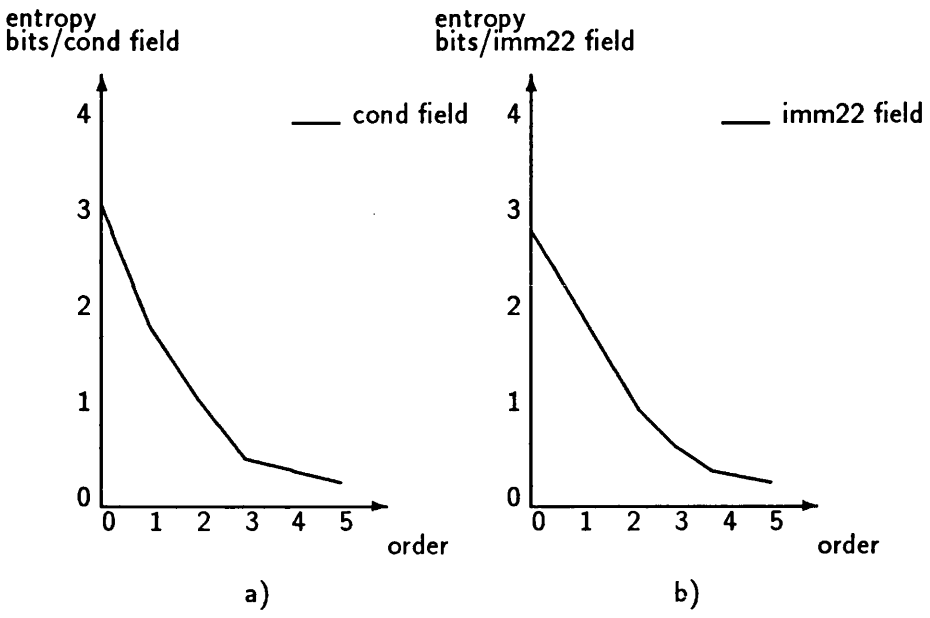

Figure 6-8: Average entropy of m-th order Markov source for the dynamic case in bits per *cond* field for case a) and bits per *imm*22 field in case b).

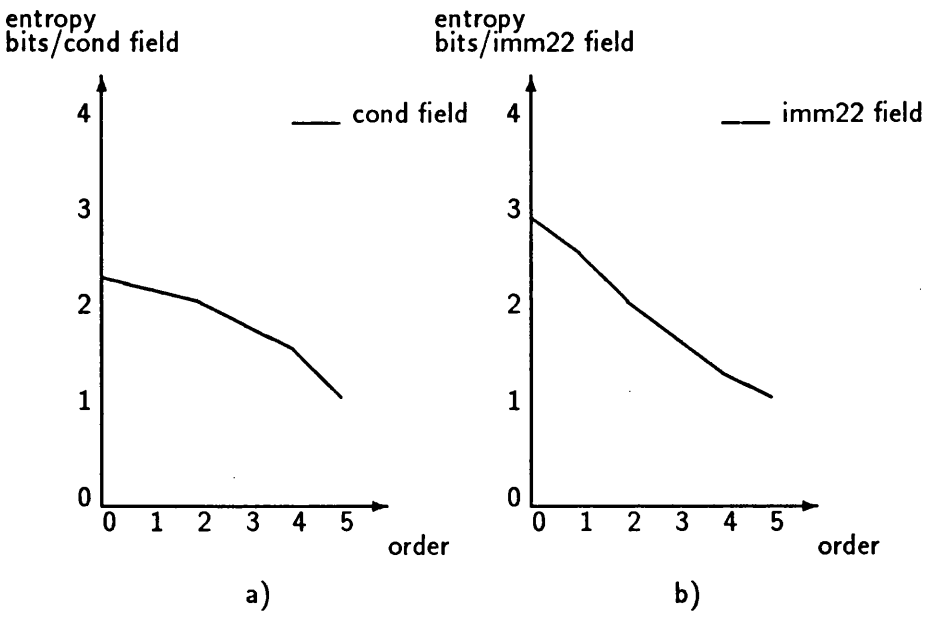

Figure 6-9: Average entropy of m-th order Markov source for the static case in bits per *cond* field for case a) and bits per *imm*22 fields in case b).

source and destination) the entropy decreases rapidly to about 0.2 bits per register field for the fifth order Markov source in the dynamic case. About the same decrease in entropy can be found for the *cond* and *imm22* bit fields.

Further experiments resulted in a variable reduction in entropy although some bit fields (such as the *asi* field) are difficult to measure as they occurred only rarely in the examples used. However, the entropy does not differ significantly from other bit fields and is therefore not printed here.

Wade and Stigall and Hammerstrom and Davidson already show what can be achieved in instruction design. But the work performed does not reduce redundancy to the amount possible according to information theory.

6.4 Coding Results

The programs used for experimental results and comparisons are described in Table 6.7. All experiments are performed on SPARC.

Program Description

name	**program description**	**size in bytes**
prog1	Simple C program with a small loop, compiled for SPARC using gcc 1.4 without optimization	24 468
prog2	Coding program written in C, encoding a C file, compiled for SPARC using gcc 1.4 without optimization	2 432 976
gperf	Perfect hash function generation for six key words, compiled for SPARC using gcc 1.4 without optimization	698 436
dhry	Dhrystone benchmark for 100 runs through Dhrystone version 2.1, C language, compiled by gcc 1.4 without using the "register" attribute and optimization	640 340
obj1	executable file for VAX, compilation of a program	21 504
obj2	executable file for Apple Macintosh, "knowledge support system program"	246 814

Table 6.7: Description of programs used for experimental results. **prog1**, **prog2**, **gperf**, and **dhry** are used for experiments on the SPARC architecture.

The experimental results with arithmetic coding on static object code, using the zero and first order Markov model, are shown in Table 6.8 [Schoepke 1992f]. Static

Static Code

name	object code *size in symbols*	entropy *order 0*	encoded *bits/symbol*	entropy *order 1*	encoded *bits/symbol*
gdb	362 112	5.65	5.64	3.91	3.84
emacs	404 656	5.77	5.77	3.92	3.87
X	269 896	6.35	6.35	4.58	4.54
stat	40 144	6.12	6.12	4.00	3.95
tcsh	184 200	5.44	5.47	3.87	3.83

Table 6.8: Encoded object code in bits per symbol with entropy in bits per symbol given for the fixed zero and first order model in static case.

entries such as symbol tables or string tables are not used to build the statistics. Only the object code to be executed is examined. As a comparison the entropy is given, which provides information about the achievable compression results with arithmetic coding, measured also in bits per symbol. In the static as well as the dynamic case the stream to be encoded is built with eight bit long symbols and a fixed model is used [Witten *et al.* 1987]. The fixed model is built by a previous run over the object program so that an exact model for the object code to be compressed is used. Allowing for experimental errors the entropy bound is achieved.

Dynamic Code

name	object code *number of symbols executed*	entropy *order 0*	encoded *bits/symbol*	entropy *order 1*	encoded *bits/symbol*
prog1	24 468	5.90	5.90	2.53	2.53
prog2	2 432 976	5.94	5.95	2.90	2.90
gperf	698 436	5.94	5.94	2.91	2.95
dhry	640 340	6.00	6.00	2.89	2.87

Table 6.9: Encoded object code in bits per symbol with entropy in bits per symbol given for the fixed zero and first order model in the dynamic case.

The results for dynamic object code, using the zero and first order Markov model, is shown in Table 6.9. Compression on dynamic object code means every symbol executed during program run time is compressed. The same symbols are used to calculate the entropy. Again (considering experimental errors) the entropy bound is achieved.

Compression Schemes

scheme	description
PPMC	[Moffat 1990], based on a method proposed by Cleary and Witten, one of the best finite-context modelling methods [Cleary & Witten 1984b]
HUFF	[Gallager 1978], adaptive Huffman coding, using an order zero model
LZFG	[Fiala & Greene 1989], a coding technique based on LZ78 family [Ziv & Lempel 1978], fast encoding and decoding, and good compression without undue storage requirements
LZB	[Bell 1987], gives generally the best compression of the LZ77 family [Ziv & Lempel 1977] of Ziv-Lempel coders
DMC	[Cornack & Horspool 1987], Dynamic Markov Compression, the only finite-state modelling method described in the literature that works fast enough to support practical text compression

Table 6.10: A brief description of some compression schemes.

Experimental Results

name	size	PPMC	HUFF	LZFG	LZB	DMC
	size in bytes			*bits/symbol*		
objl	21 504	3.76	6.06	4.08	4.26	4.56
obj2	246 814	2.69	6.30	2.96	3.14	3.06

Table 6.11: Experimental results published by Bell et.al. given in bits per symbol.

Table 6.11 provides results of other coding techniques published in [Bell *et al.* 1989], also given in bits per symbol. The schemes mentioned are described briefly in Table 6.10.

6.5 Summary

This type of data provides the information required to optimize the design of a computer for the execution of programs or building an instruction set. The statistics show that there is a certain number of pairs which can be used to build a new opcode, but the benefit of an additional opcode has yet to be investigated [Bennett & Smith 1989].

The experimental analysis shows great redundancy in the instruction stream as most of the executable code is built with only a small number of opcodes and constants and branches use mainly a small number of bits. These occur with considerable regularity, reflected in the low entropy when treated as higher order Markov source. This implies that highly dense instruction streams are possible.

In arithmetic coding a suitable coding technique for encoding and decoding an instruction stream efficiently and effectively has been found. With this technique encoding as close as desired to the entropy is achievable, and the execution of compressed object code is possible [Schoepke 1992a].

Chapter 7

Proposed Model for a Compressed Instruction Set Architecture based on SPARC

The experiments carried out show it is possible to generate high density instruction streams, especially when treated as higher order Markov source because of dependencies between instructions. Because of high redundancies in instruction streams very low entropy can be reached and encoding into less than three bits per symbol is possible using only a first order model. This is a reduction of nearly 70%.

In this chapter a model is proposed to execute such dense object code and to implement the system.

7.1 Standard Execution Model

SPARC is characterized by the following concepts:

1. The main units are a control unit, arithmetic and logic unit, a memory, and input and output facilities.

2. Programs and data share the same memory, thus the concept of a stored program is fundamental.

3. The control and arithmetic units, usually combined into a central processor, determine the actions to be carried out by reading instructions from the memory.

It follows that a program consists of a set of instructions that are examined one after another; a program counter (PC) in the control unit indicates the next location in memory from which an instruction is to be taken.

The three distinct phases that constitute the sequencing of each instruction for this architecture are

1. Determining the memory address which contains the instruction,
2. Fetching the instruction from memory and
3. Executing the instruction.

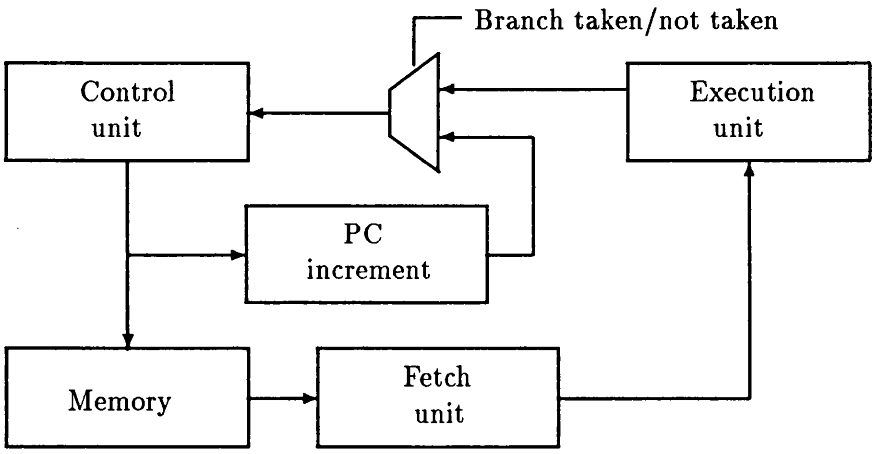

Figure 7-1: Instruction execution: Instructions are fetched from main memory into the fetch unit and provided to the execution unit to execute. If a branch has been taken, the new memory location has to be calculated. If no branch has been taken, the PC is incremented to the next instruction.

Before a processor can execute an instruction it must fetch the instruction from memory. Before this operation can occur, the instruction pointer must be updated. If a branch has been taken, the address of the next instruction to be executed has to be calculated. Therefore the rate at which the processor executes instructions cannot exceed the rate at which instructions are fetched from memory. During the execution phase of each instruction, the processor determines the memory location of the next instruction to be executed. Figure 7-1 shows an example of a von Neumann architecture.

7.2 Decode-Execution Model for Compressed Code

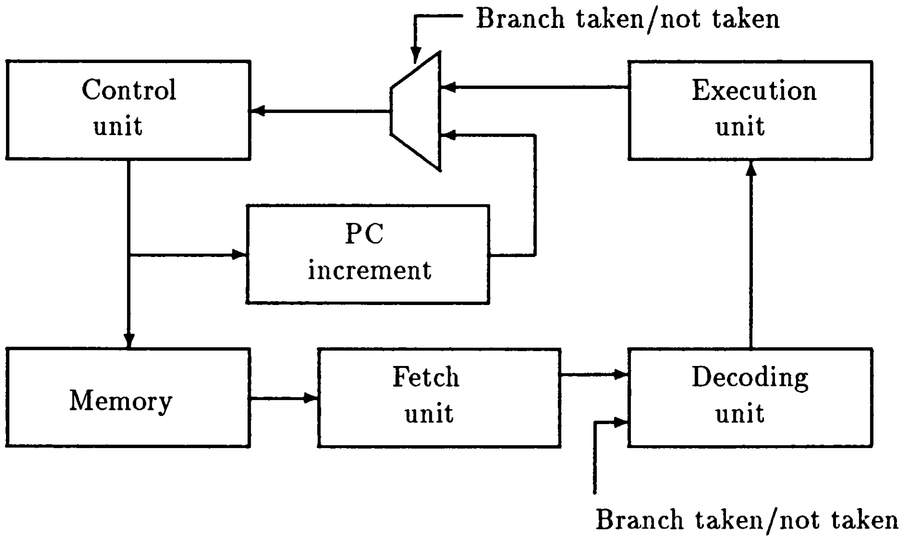

Figure 7-2: Execute Compressed Object Code: Chunks are fetched from main memory into the fetch unit and decoded in the decoding unit. Decoded instructions are provided to the execution unit to execute. If a branch has been taken, the new memory location has to be calculated and the decoder has to be re-initialized. If no branch has been taken, the PC is incremented to the next instruction.

To execute compressed object code is more complicated than for conventional uncompressed object code. Before executing an instruction it is necessary to fetch new information, the *chunks*, from the main memory or RAM through the fetch unit into the decoder, only if required. The term *chunk* is used to describe eight bits of the encoded bit stream. Operating independently as far as possible from the execution unit the fetch unit fetches four chunks from main memory which stores the dense instruction stream. The size 32 bits is chosen as no change in the fetch width of the processor is made. The decoding unit decodes the fetched chunk, builds the instruction, and provides it for the execution unit to read. Having done that, the execution unit is able to execute the instruction after reading it from the decoding unit.

Because of the dense instruction stream (provided by the main memory), it is not always necessary to fetch new chunks for the decoder after executing one or more instructions. In the worst case, i.e. after taking control transfers, new chunks are required for the decoder, but in the best case, several instructions can be executed without another memory access.

Figure 7-2 gives an example of a suggested architecture to execute compressed object code with only one instruction stream to be decoded. For decoding several streams reflecting the structure (i.e. different fields) given in an instruction stream, the given architecture has to be multiplied by the number of streams to be coded including an on-chip connection between the decoding units for communication. Fetch and decoding units can operate most of the time independently of the execution unit, unless the flow of control is altered. Therefore the decoding unit can decode the next instruction in advance if possible. Fetch, decoding, and execution units are assumed to be on one chip.

The changes in the architecture proposed compared to the existing architecture are kept to a minimum – no changes in the instruction set executed are made.

7.3 Handling Control Transfers

One of the most difficult problems is how to handle control transfers, because control transfer instructions not only alter the control flow of the program from sequential execution. They also effect the whole encoding and decoding phase as encoding has to be on static code and decoding on dynamic code. Several provisions are necessary to handle these cases correctly.

7.3.1 Overview of the Problem

Like sequencing instructions [Krick & Dollas 1991], compressed object code execution suffers after having taken control transfers, because the new memory address has to be calculated and new chunks are necessary to continue decoding the dense instruction stream. Furthermore, one cannot rely on information further back and therefore the context is lost. This influences the compression results on both the static and dynamic code when higher order models are in use. The coding algorithm has to be restarted with only little or no knowledge of the context and at an addressable memory location.

```
if ( i == 0 )
    i++;
else
    i--;
```

```
for ( i = 0; i <= 3; i++ )
{
    j++;
}
```

Figure 7-3: C Code for the *if* and *for* Statement

```
    ld   [%fp-20], %o0
    tst  %o0
    bne  L1
    nop
    ld   [%fp-20], %o0
    add  %o0, 1, %o0
    st   %o0, [%fp-20]
    b    L2
    nop
L1:
    ld   [%fp-20], %o0
    add  %o0, -1, %o0
    st   %o0, [%fp-20]
L2:
```

a) *if* statement

```
    st   %g0, [%fp-20]
L3:
    ld   [%fp-20], %o0
    cmp  %o0, 3
    bg   L4
    nop
    ld   [%fp-28], %o0
    add  %o0, 1, %o0
    st   %o0, [%fp-28]
    ld   [%fp-20], %o0
    add  %o0, 1, %o0
    st   %o0, [%fp-20]
    b    L3
    nop
L4:
```

b) *for* statement

Figure 7-4: SPARC assembler language for statements in Figure 7-3

Figure 7-3 provides C code for an *if* statement in case a) and for a *for* statement in case b). The SPARC assembler language for both statements is provided in Figure 7-4 (generated by the gcc compiler without optimization [Stallman 1989]). For case a) the compiler generates a conditional branch forward (after the *if* condition) and an unconditional branch forward (after statement *i++*). For case b) a conditional branch forward (after the loop condition has been examined) and an unconditional branch (after statement *j++* and the loop conditions) has been generated. These two examples demonstrate the frequency of control tranfers for two common C HLL statements and emphasize the problems in handling branches (and control transfers) effectively and efficiently.

Assume the first encoding phase starts at the first ld of the *if* statement. The encoder can encode the instruction stream until it reaches the L1 label. At this point the encoding process is interrupted and the second encoding phase has to be started because L1 can be an entry point during program run time. The second encoding phase is interrupted at the second label L2 and so on. However, the number of instructions to be encoded for each coding phase depends on the compiler used, the application to encode, and the target architecture. The influences on the compression results have not been investigated. In the *for* statement case L3 and L4 interrupt the encoding process.

Branch	**Frequency**
static	12.25%
dynamic	16.30%

Table 7.1: The average branch frequency for static and dynamic cases is shown. The data was collected for the SPARC architecture.

Table 7.1 shows the branch frequency for the static as well as the dynamic case collected for the SPARC machine. In the static case about every eighth instruction is a branch instruction. In the dynamic case branches are more frequent, about every sixth instruction is a branch instruction.

The detection of branch targets can be done by the compiler as the compiler "knows" these entry points. It then can pass the information on to the encoder. It may, however, turn out that destinations of indirect jumps are undetectable if the user decides the jump destination during program run time.

The scheme of "windowing" instructions is based on the observation that many data files exhibit a strong locality behaviour of references. In the example of DISC, a "window" is defined as a group of instructions which are issued to functional units for execution at the same cycle [Wang & Wu 1991]. Entry into a window takes place only at the leader, and exit from a window can occur at any instruction. A leader is defined as either the first instruction of the program or any branch–target instruction. The size of the window, or the number of instructions in the window, depends on the number of functional units in the system.

The window is therefore similar to a "basic block" in source-language code. However, a basic block can only be left from the last instruction. Thus, an instruction following a branch instruction cannot reside in the same basic block as the branch. This constraint is not imposed in the window. This means that a window can contain multiple basic blocks. In windows all instructions between branch instruction and branch target form the base for one coding step. The disadvantage from the compression point of view is the loss of context between windows and therefore the loss of compression.

7.3.2 Compiler Information

Detecting entry points can be done by the compiler. This involves branch instructions and function calls with fixed displacement, the jump and link instruction, and trap instructions. However, the destination of register indirect jumps is difficult to calculate if not used for function returns. It is not possible to detect all control transfer destinations since changes caused by the user cannot be predicted or calculated in advance.

7.3.3 Solutions

Initialize Coder with Simple Context

Reinitialize the decoder after a branch has been taken by clearing all coding information is a simple way to solve the problem. On the one hand we lose all the context information and therefore get worse compression results. On the other hand this method is considerably faster due to fewer memory accesses and it is simpler to implement. Furthermore, no memory is required to store context information.

Reinitialize Decoder with Context Information

One way to avoid loss of context during coding is to store all context information at each branch target address. After a branch has been taken, the coder can be reinitialized with all the relevant context information required to continue coding. The great advantage is one does not suffer from loss of structure and therefore one can reach higher compression. On the other hand it is very time and memory expensive to store all necessary context information at each branch target. Consider the gdb program with 90528 instructions. If every eighth instruction is a destination address the amount of memory needed to store the context information for a first order model is about 11kbyte. Since different applications use different destinations it is necessary to load the correct table with each application. Therefore this solution is not practicable.

Variable Length Context

With variable length contexts one can solve many of the above problems. After a branch has been taken, one starts the decoding phase with no information about the instruction stream seen so far, i.e. a zero–order model. After the first symbol has been coded, one can use a higher order model, i.e. the first–order model. Continuing in this fashion one can get a higher order after each symbol has been coded until a control transfer instruction alters the control flow. The advantage is that one loses not all of the structure given in an instruction stream after a control transfer has been taken and that one can reach high order after coding a few instructions. The disadvantage is that during decoding one has to change the order to keep in step with the encoder.

Execute Graphs

Static object code, on which the encoding process has to be done, is provided in linear order. In contrast, looking at dynamic code (on which decoding has to be done), it is more like executing a graph than executing linear code. During program execution the linearity is broken by branches taken or subroutine called.

To achieve very high compression it is necessary to know the context of each instruction to be encoded. This problem could be solved by transforming linear code into a graph representing executable code. At each branch target address one splits the code into two parts: the "linear" part which represents the non taken branches and the "graph" part which represents the taken branches. During program run time one now gets different context for the linear part than for the graph part. The great advantage is that compression can be done with an ideal model, as the context is always adequate. The great disadvantage is that one has to store several object code parts in graph mode for one part in the original linear code. This makes the solution probably unworkable in practice as the increase in static code size would be tremendous. This method has yet to be investigated.

7.3.4 Method Applied

The influence of control transfer instructions on the dense instruction set architecture should not be underestimated. Reducing static code size depends not only on the model used but also on the method used on how to handle entry points effectively.

The method used has to be balanced carefully against the compression ratio achievable (especially with higher order Markov sources), the amount of memory required to store the context information, and the time required by the method used for decoding.

Due to the complexity of designing and implementing different methods in the time available the simplest method described was implemented, initializing the decoder with a simple context.

7.4 Algorithms for Encoding and Decoding

The encoding process must be preceded by post compiling to add information necessary to decode the given instruction stream. The algorithms described in Figure 7-5 [Schoepke 1992b] give a brief overview how post compiling has to be done to detect control transfer instructions during the encoding process and how the decoder can examine the dense instruction stream to provide instructions to the execution unit for correct execution.

A new phase of encoding has to be started at each entry point. This includes ending the current encoding phase through outputting remaining bits from the encoder, inserting an additional decode instruction so that the decoder is able to recognize the entry points, and initializing the encoder for further encoding. This influences the achievable compression results especially as one knows that only a few instructions

```
Encoding_Program ()
{
   Initialize_Arithmetic_Encoder ();
   while !( EOF )
   {
       read ( instruction );
       if ( instruction == entry_point )
          Encoding_Procedure ( instruction );
       Encode_Instruction ( instruction );
    }
}

Encoding_Procedure ( instruction )
{
   End_Current_Arithmetic_Encoding_Phase();
   Insert_Decode_Instruction ();
   Initialize_Arithmetic_Encoder ();
   return;
}
```

Figure 7-5: Encoding algorithm

are between branch instruction and branch target. After initializing the encoder, the instruction can be encoded.

The decoding algorithm (Figure 7-6) shows a brief summary of the decoding and execution phase. The decoder gets one chunk from main memory, decodes it, and supplies the symbol to build the instruction. Building an instruction depends heavily on the number of instruction streams, the size of the symbol to be decoded and the type of the instruction to be built. Therefore several steps could be necessary to build an instruction. If the instruction is a decode instruction, the decoder has to be re-initialized. The same is necessary in the case of a branch taken where the new memory address also has to be calculated.

7.5 Cache Influence

The cache organization for the proposed architecture can be done in several different ways.

```
Decoding_Program ()
{
  Initialize_Arithmetic_Decoder ();
  read ( chunk );
  while !( program_end )
  {
     Decode_Symbol ( chunk );
     Build_Instruction ( symbol );
     if ( instruction == decode_instruction )
       Initialize_Arithmetic_Decoder ();
     else
       if ( instruction == branch_instruction )
         Decode_Procedure ( instruction );
      else
         Execute_Instruction ( instruction );
    read ( chunk );
  }
}

Decoding_Procedure ( instruction )
{
  Execute_Branch_Instruction ();
  if ( branch_taken )
  {
    Initialize_Arithmetic_Decoder ();
    Determine_New_Address ();
  }
  return;
}
```

Figure 7-6: Decoding algorithm

1. Fetched chunks from memory can be stored in the on-chip cache. The advantage is that several chunks are available without another memory reference except after taken control transfers. The disadvantage is that these chunks have to be decoded before execution. Therefore this is not a desirable solution.

2. Decoded instructions can be stored in the on-chip cache. The advantage is that the decoding is already done and the instruction can be read by the execution unit ready for execution like in any similar architecture. The disadvantage is that more cache memory is needed to store the decoded instruction than the chunks.

3. Fetched chunks, as well as decoded instructions, can be stored in the on-chip cache. This solution combines some of the advantages and disadvantages of the previous ones, e.g. availability of instructions for the execution unit.

Furthermore, the proposed architecture would gain from a second level cache that could hold a large number of encoded chunks as dense object code size is much smaller than uncompressed object code. How different solutions improve system performance and their influence on compressed code has yet to be investigated.

7.6 Hardware Design

A high level hardware design example for a VLSI chip to implement dense instruction set computers using four decoders [Schoepke 1992g] is given in Figure 7-7. The architecture proposed is based on a decoder with eight bit input and output. The whole decoding unit is assumed to be on one chip. During execution the fetch unit fetches information from memory 32 bit wide into a fetch buffer and provides eight bit long chunks to the decoder as required. The decoders decode the chunks and provide each eight bits to build the instruction which is transferred to the CPU for execution. The CPU has to acknowledge the decoder after a control transfer has been taken or a block entry point has been reached to reset the decoder. The decoding process can be done in parallel to the instruction execution until the control flow is altered. In this particular example four decoders are used.

Figure 7-8 shows a design to implement arithmetic coding [Schoepke 1992g]. The decoder has eight bit input and output, a connection to the memory containing probability information, and a connection to the CPU providing information about control transfers and their targets. The address decoder, using *Current C* (the current lower coding point) and *Current A* (the current interval width), calculates the address which is used by the symbol finder to get the symbol from RAM. The symbol is then provided to build the instruction. In parallel to other CPU activities the decoder calculates the values for *New C* and *New A* in two different cycles. The new low point *New C* of the next interval is the sum of the current low point *Current C*, and the product of the

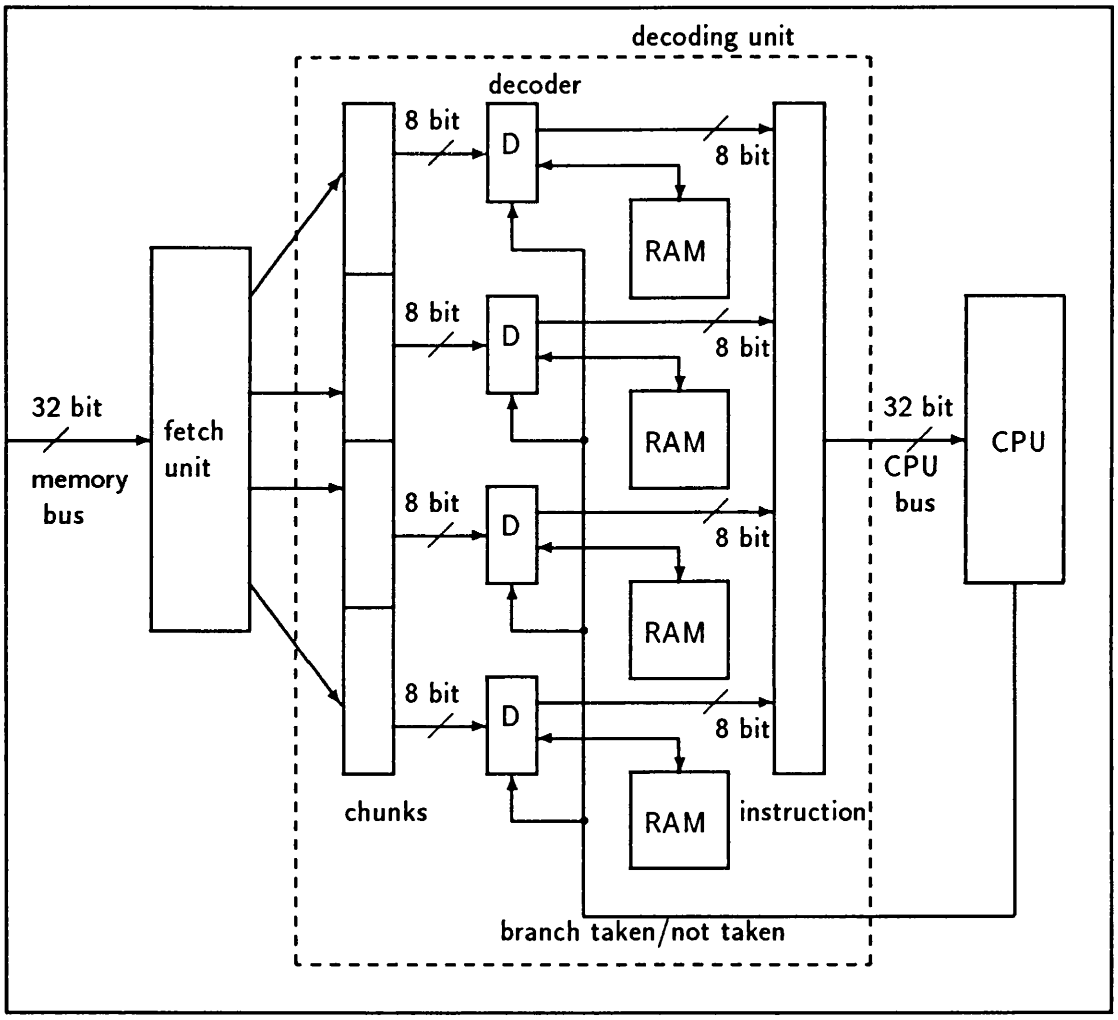

Figure 7-7: VLSI design for the dense instruction set computer architecture with four decoders. The fetch width from memory is 32 bits. The fetch unit provides chunks of eight bit to the decoding unit as required. The decoders supply eight bit each to build the 32 bit instruction. The decoding unit provides the instruction to the CPU for execution.

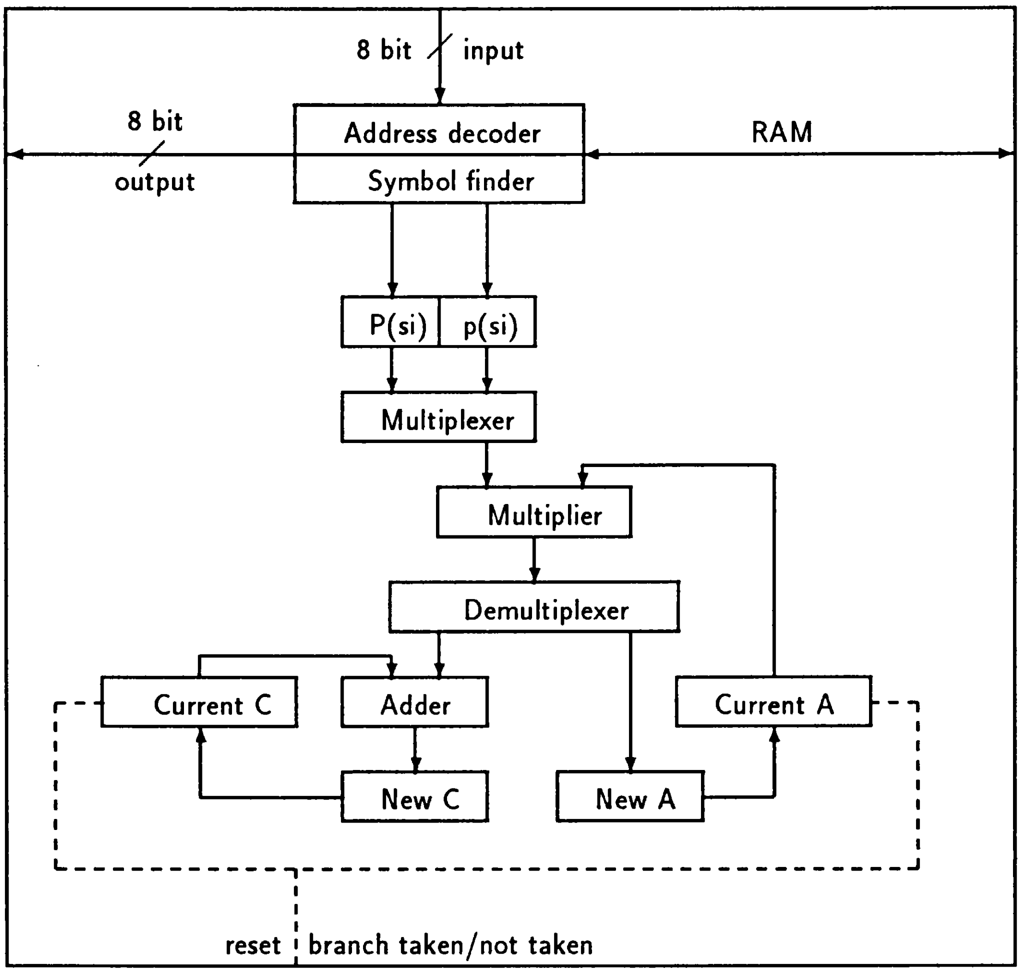

Figure 7-8: Hardware design for arithmetic coding. The arithmetic decoder takes an eight bit input (the chunk), decodes it, and supplies an eight bit output to build the instruction. The arithmetic decoder uses the address decoder and the symbol finder for decoding. The CPU connection reinitializes the current lower point, *Current C*, and the interval width, *Current A*, if necessary.

current interval width *Current A* and the cumulative probability $P(s_i)$ of the symbol s_i to be encoded:

$$New\ C = Current\ C + (Current\ A * P(s_i)) \tag{7.1}$$

The new interval width *New A* is a product of the current interval *Current A* and the probability $p(s_i)$ of the symbol s_i to be encoded:

$$New\ A = Current\ A * p(s_i) \tag{7.2}$$

The calculation is as follows: In the first cycle the product from *Current A* multiplied by the cumulative probability $P(s_i)$ has to be calculated and in the second cycle the decoder calculates the product *Current A* multipied by the probability $p(s_i)$ and the sum *Current C* plus the product *Current A* multiplied by the cumulative probability $P(s_i)$, which has been done in the first cycle. Assuming a cycle time for this stage only double the speed of the processor cycle time, the calculation finished at the same time the CPU has executed a one cycle instruction.

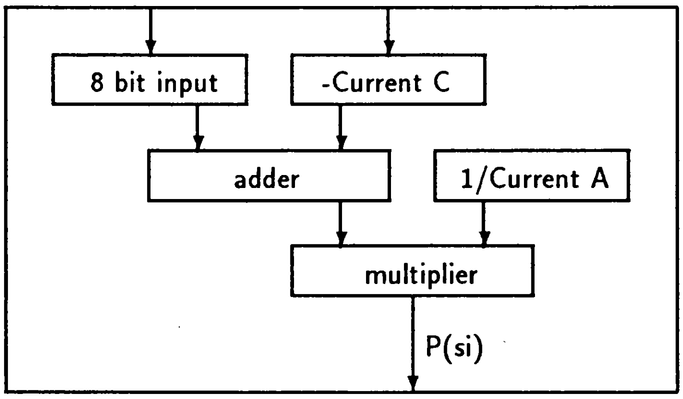

Figure 7-9: The diagram shows the hardware design for the address decoder. The address decoder needs the current lower point C, the current interval width A, and the eight bit long input to calculate the cumulative probability, $P(s_i)$. The quantity $P(s_i)$ is supplied to the symbol finder.

A high level hardware design for the address decoder [Schoepke 1992e] is given in

Figure 7-9. The address has to be calculated as follows:

$$address = (8\_bit\_input - Current\ C)/Current\ A \qquad (7.3)$$

The resulting address is a close one, but not the exact address one is looking for. A symbol finder (Figure 7-10) [Schoepke 1992e] therefore has to search in the cumulative probability table for a symbol with an address equal or less to the address calculated. It then provides the probability and cumulative probability of the symbol to the arithmetic coding unit. Using the enhanced algorithm provided in Section 5.5 finding a symbol in the cumulative frequency table reduces to one comparison.

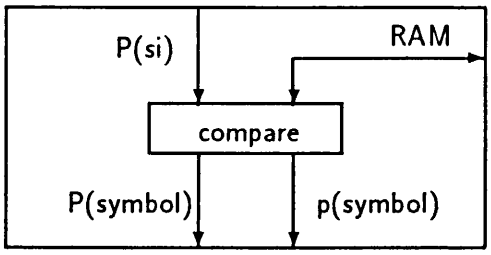

Figure 7-10: The hardware design for the symbol finder is shown. The symbol finder takes the cumulative probability, $P(s_i)$, supplied by the address decoder and compares it with the information provided in the RAM. It then delivers the probability of the symbol, $p(symbol)$, and the cumulative probability of the symbol, $P(symbol)$, to the arithmetic decoder.

The whole process of decoding an instruction has to be done as far as possible in parallel to the instruction execution as real time decoding is necessary. Assuming the current environment of a 25MHz SPARC station and 100MHz adder/multiplier, one is able to decode an instruction in half the CPU cycle time. The address calculation can also be done with the same speed, but some calculations have to be done sequentially. Increasing processor clock rates only would, of course, change this ratio.

With the decoding algorithm developed the symbol finder can be implemented with one comparison only considering a look-up table of size four kbyte for a zero order model. For higher order models the look-up table size increases.

In parallel to instruction execution *New C* and *New A* can be calculated but both values are needed in the address decoder to decode the next symbol. Therefore faster units are needed to decode in real time. Furthermore, precautions have to be taken to deal with the under- and overflow problems.

7.7 Summary

A description of the dense instruction set architecture and the algorithms used is given. The proposed decode–execution model seen in Section 7.2 is able to handle the decoding phase. With arithmetic coding and a fixed model it is possible to handle control transfers efficiently.

Chapter 8

Dense Instruction Set Computer Architecture Simulation

In this chapter I analyze the results obtained by simulation of the architecture given in Chapter 7, show the limits of the work done, and discuss the differences between theory and the results.

8.1 Encoding and Decoding Process

During the encoding phase on the static object code, entry points are detected and the instruction stream is compressed. During decoding on the dense dynamic object code the instruction stream is de-compressed.

8.1.1 Encoding

Entry Points

The encoding process is preceded by post-compiling to get the information necessary to encode and decode the instruction stream. The object code has to be examined for branch targets, function calls and returns, and jump and link instructions as well as trap instructions. This information must be passed on to the encoding unit. Detecting branch targets, whether unconditional or conditional, can be done by reading the *disp22*

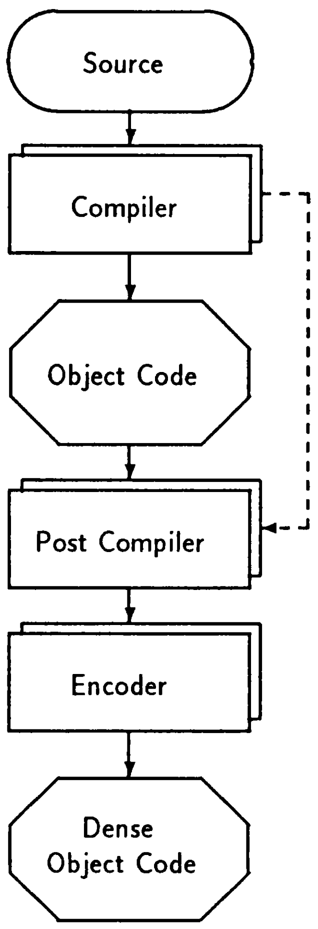

Figure 8-1: The encoding graph is shown. The compiler compiles source code to object code and supplies information to the encoder about entry points. This information is used during post-compiling of the object code. Then the encoder encodes the object code to the dense object code.

bit field from the branch instruction. The destination is then given by the program counter plus *disp22* multiplied by four. During the encoding phase the program counter is initialized to zero. The target address found is an entry point where the arithmetic encoding (and decoding) process has to be restarted. This involves a re-initialization of the coding range with lower and upper bound (Chapter 5) and ending of the current coding phase, which involves outputting the remaining bits for the encoder. If a higher order model is in use, the context of the next symbol to be encoded or decoded must be initialized.

The same method of finding the entry point has been used for *call* instructions. The *disp30* bit field multiplied by four provides the target address (the program counter is again initialized to zero). The target address found is again an entry point where the same actions are necessary to encode and decode the instruction stream.

For function returns (a register indirect jump and link instruction) finding the right target address is more complicated as the address can change during program run time. This information cannot be collected by examining just the object code, since register contents and thus return address can change. Therefore the current architecture is restricted. Changing the return address during run time to an unknown entry point, i.e. in between two entry points, is not tolerated. Using the jump and link instruction in such manner results in a program run time error. Changing the address to any other entry point (such as another procedure return address) is possible as these points are already treated as entry points.

During simulation it became clear that not all addresses created by the jump and link instruction within the examples analyzed are known entry points. In particular, some library routines linked into the object code use addresses that are not known to the encoder so far. The restrictions imposed can be abolished when compiler information is available to the encoder regarding addresses used by control transfer instructions.

Encoding Model

Symbol context is needed to encode (and decode) with higher order models. The information concerned could be stored in on-chip RAMs but it was found to be an unacceptable overhead since the context could be different for each entry point. Instead, common context information has been used. The loss of context and re-initialization of the coder influences the compression results and the results achieved during program execution as will be seen.

Encoding Process

The encoding process depends on the model used and the technique chosen to encode the instruction stream. The simulator divides the 32 bit long instruction into four eight bit symbols. The encoder encodes each eight bit symbol separately in the order of their occurrence. That means the last symbol of the previous instruction is the context information for the first symbol in the current instruction. If the fetch unit fetches an instruction from a location that is an entry point, the encoder puts out the remaining bits and re-initializes the coding range via lower and upper bounds. As the fetch unit during decoding uses the same 32 bit alignment during program run time as the original architecture, the encoder must ensure that the first byte that is put out by the encoding unit (the *chunk*) is on a 32 bit alignment. This alignment leads to wasted bits that fill the gap between the last chunk written by the encoder and the 32 bit alignment.

The encoding process ends after all instructions have been encoded. As it was not always possible to distinguish between data and instructions with the method used (examining object code only), data that could not be distinguished is also encoded. This is of no importance during simulation as the data references are done on the original program. The data that can be recognized (such as data from the data segment) was not encoded.

8.1.2 Decoding

The decoding process is time critical as it has to be done during program run time. The program is loaded into main memory. During program execution the fetch unit fetches information from main memory in 32 bit blocks, i.e. four chunks. The decoders decode the information provided by the fetch unit and build an executable instruction for the execution unit. The instruction then can be executed. The process is straightforward until a control transfer instruction alters the control flow or an entry point has been reached. Coming into this stage involves further actions other than just decoding and executing an instruction. Reaching an entry point means re-initialization for the decoder, and restarting the decoding process with a known range to remain in step with the encoder. The new fetch address has to be calculated.

8.2 A Simulator for Dense Instruction Set Architectures

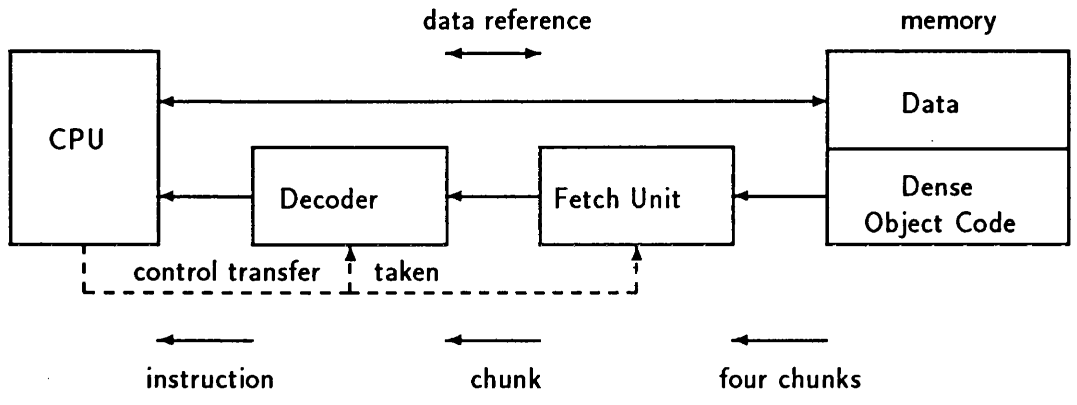

Figure 8-2: The simulator for decoding and instruction execution. The data path between memory and CPU has not been changed. On the instruction path four chunks are fetched from memory (supplying dense object code) and stored in a fetch buffer 32 bits wide. One chunk is fed in the decoder. The decoder output is stored in an instruction buffer until building of the instruction is completed. The instruction is then supplied to the CPU.

The dense instruction set architecture was simulated on the SPARC machine using two child processes traced by a parent process. For the first process main memory supplies a stream of four chunks from the dense object code to the processor fetch unit. The fetch unit transfers one chunk to the decoder if required. This technique requires a buffer of 32 bits in the fetch unit. Between fetch and execution unit the decoder decodes the encoded instruction stream and builds the instruction that is fed into the execution unit. The instruction supplied to the execution unit has to be 32 bits wide as there is no change in the instruction set finally executed on the machine. In the execution unit the instruction is executed.

In the simulation system one decoder decodes every chunk sent by the fetch unit. Therefore, one decoder has to transmit four bytes as four bytes are necessary to build an instruction 32 bits wide.

Executing a control transfer instruction or reaching an entry point involves the action of re-initializing the decoder, and starting with new coding range as mentioned before. New chunks have to be supplied to the fetch unit from an addressable memory location before decoding can continue. The symbol context has to be initialized.

A further problem exists in how to recognize an entry point during run time which is not preceded by a taken control transfer. The simulator uses a table of entry points, built during encoding, to detect them.

Some compilers use space between blocks of compiled code to store data. This can result in smaller code size, but makes analyzing object code an impossible task as data cannot be distinguished from instructions. The simulator implemented therefore uses two different environments (running in two different child processes) and is thus able to deal with instructions as well as data references. One environment (supplying chunks) simulates the fetch, decode, and execution unit, i.e. it can handle all instruction references. The second environment (supplying the data segment) deals with data references. As a result, instructions are fetched from the encoded program, data from the original program. As there are no changes in the data address space, the program executes correctly.

8.3 Simulation Results

The results are based on simulations done on SPARC, using the same object programs as used during experiments with arithmetic coding. During simulation no on-chip cache was used.

8.3.1 Compression Achieved

Table 8.1 shows the compression results obtained by the four programs using a zero order model. The average reduction is about one bit per symbol which results in an average decrease in program size of about 12%.

Table 8.2 shows the compression results obtained by the four programs using a first order model. The average reduction achieved is more than three bits per symbol or about 44%.

Compression Results
Zero Order Model

program	**achieved**	**reduction**
	bits/symbol	*percent*
prog1	7.06	11.75%
prog2	6.81	14.88%
gperf	7.13	10.88%
dhry	7.10	11.25%
average	7.03	12.19%

Table 8.1: The compression results using a zero order model are shown. The information is given in bits per symbol and in percentage of reduction.

Compression Results
First Order Model

program	**achieved**	**reduction**
	bits/symbol	*percent*
prog1	4.48	44.00%
prog2	4.00	50.00%
gperf	4.53	43.50%
dhry	4.97	37.88%
average	4.50	43.85%

Table 8.2: The compression results using a first order model are shown. The information is given in bits per symbol and in percentage of reduction.

8.3.2 Compression Results compared to Theory

Table 8.3 shows the compression results compared to the theoretically possible entropy using a zero order model. On average the compression result achieved is about one bit higher than the entropy bound.

Table 8.4 shows the compression results compared to the theoretically possible entropy using a first order model. On average the compression result achieved is about 1.7 bits higher than the entropy bound.

In both cases, it is not possible to achieve the optimum entropy as the compression results are largely affected by control transfers as encoding has to be done on static code and decoding on dynamic code. Another factor is the model used, since one model

Compression Results compared to Entropy
Zero Order Model

program	**entropy**	**achieved**
	bits/symbol	*bits/symbol*
prog1	5.90	7.06
prog2	5.94	6.81
gperf	5.93	7.13
dhry	6.00	7.10
average	5.94	7.03

Table 8.3: The compression results using a zero order model are shown. The information is given in bits per symbol.

Compression Results compared to Entropy
First Order Model

program	**entropy**	**achieved**
	bits/symbol	*bits/symbol*
prog1	2.53	4.48
prog2	2.90	4.00
gperf	2.96	4.53
dhry	2.89	4.97
average	2.82	4.50

Table 8.4: The compression results using a first order model are shown. The information is given in bits per symbol.

cannot represent all applications.

8.3.3 Comparison between Zero and First Order Model

Table 8.5 shows the entropy results comparing zero and first order model. It illustrates the improvement achieved using a context dependent model.

Table 8.6 shows the compression results comparing zero and first order model and illustrates the improvement achieved using a context dependent model, i.e. a first order model, rather than an independent model, i.e. a zero order model. The improvement achieved is on average 17% less than the entropy suggests.

Entropy Improvement
Zero and First Order Model

program	**entropy**		**improvement**
	zero order	**first order**	
	bits/symbol	*bits/symbol*	*percent*
progl	5.90	2.53	57.12%
prog2	5.94	2.90	51.18%
gperf	5.93	2.96	50.08%
dhry	6.00	2.89	51.83%
average	5.94	2.82	52.53%

Table 8.5: The entropy results comparing zero and first order model are shown. The information is given in bits per symbol for zero and first order model and in percentage of improvement from zero to first order model.

Compression Improvement
Zero and First Order Model

program	**achieved**		**improvement**
	zero order	**first order**	
	bits/symbol	*bits/symbol*	*percent*
progl	7.06	4.48	36.54%
prog2	6.81	4.00	41.26%
gperf	7.13	4.53	36.47%
dhry	7.10	4.97	30.00%
average	7.03	4.50	35.99%

Table 8.6: The compression results comparing zero and first order model are shown. The information is given in bits per symbol for zero and first order model and in percentage of improvement from zero to first order model.

8.3.4 Effectiveness

Table 8.7 shows the effectiveness of the dense architecture. The results gathered by simulation are compared to the entropy bound.

For the zero order model a decrease of about 26% is possible and a reduction of about 12% has been achieved. This results in an effectiveness of 47.09%. For the first order model a reduction of 65% is possible. During simulation a decrease of 44% has been achieved. This results in an efficiency of 67.57%. The results are better for the first order model as a context dependent model is able to represent the code more accurately.

Effectiveness

	zero order	**first order**
entropy	5.94 *bits/symbol*	2.82 *bits/symbol*
achieved	7.03 *bits/symbol*	4.50 *bits/symbol*
effectiveness	47.09%	67.57%

Table 8.7: Effectiveness

8.3.5 The Effect of Fetch Width

Table 8.8 provides the information on how many bits are wasted using 32 bit fetch width against eight and one bit fetch width for the zero order model.

Average Percentage of Wasted Bits
Zero Order Model

fetch width	**wasted bits**
bits	*percent*
1 bit	0%
8 bits	3.01%
32 bits	14.18%

Table 8.8: The average percentage of wasted bits by 32 bit fetch width compared to eight bit fetch and one bit fetch width using a zero order model is given.

Using eight bit fetch width only 3.01% of bits is wasted. Using 32 bit fetch width, however, already one out of eight bits is wasted.

Table 8.9 provides the information on how many bits are wasted using 32 bit fetch width against eight and one bit fetch width for the first order model.

As in the previous case only a small number of bits is wasted using eight bit fetch width instead of one bit fetch width. Using 32 bit fetch width this increases to nearly two out of eight bits.

Comparing both tables the increase for eight bit fetch width from zero to frst order is about 22%. For 32 bit fetch width the increase is about 59%.

Average Percentage of Wasted Bits
First Order Model

fetch width	**wasted bits**
bits	*percent*
1 bit	0%
8 bits	3.67%
32 bits	22.62%

Table 8.9: The average percentage of wasted bits by 32 bit fetch compared to eight bit fetch and one bit fetch width using a first order model is given.

8.4 Architecture Limits

In the current state the simulated dense architecture has some limits. The simulation used is concerned with single tasks only, using a simple model for encoding and decoding. Applying the solution to a more general architecture involves further steps which have not yet been investigated.

Building the architecture proposed one has to consider task changes, interrupts and other mechanisms which can occur. Task changes, for example, require saving the coder status (this is lower and upper bound for the decoder and the bits left in the fetch unit). Therefore, task changes become more costly for the dense architecture than for the conventional architecture.

Analyzing object code only is not sufficient to build a dense instruction set as not every entry point can be calculated. That means compiler changes are necessary to deal with register indirect control transfers. The experience gained shows that most of the time the entry point is already known, but the method cannot deal with all applications and compilers or languages.

Self-modifying code is not supported on the dense architecture as the coding process is based on streams of symbols rather than individual symbols.

8.5 Summary

The dense architecture has been simulated on the SPARC machine. The post-compiling process on the object code to compress provides the necessary information about entry points (a minor number of entry points had to be treated separately). The encoder then encodes the instruction stream considering a specific model. During program run time the decoder decodes the dense instruction stream using the same model. Reaching an entry point, the decoder re-initializes the coding ranges and fetches new chunks from memory.

The compression results show a substantial reduction in instruction bandwidth, especially with a first order model. Comparing the results to the entropy bound the effectiveness achieved approaches 68%. The percentage of bits wasted using 32 bit fetch width instead of one bit fetch width reaches 23%.

Chapter 9

Conclusion

In this dissertation I present one direction to improve system performance (especially for multiprocessors with shared memory) using dense computer instruction sets. This proposal has certain advantages over existing approaches. I have shown it is possible to reduce memory-processor bus traffic by a factor of about two on the SPARC architecture with a simple first order model. This work is not complete as necessary compiler modifications are not yet implemented. Further reduction in memory-processor bus traffic can be achieved with more sophisticated models which exploit the structure of instruction streams. This is, however, a task of considerable proportion going beyond the frame of this dissertation. Still, this dissertation opens up an avenue for further research into instruction set compression.

There is no reason why the dense instruction set computer architecture cannot be built in VLSI. Despite the changes necessary to accommodate various requirements in different architectures (such as different instruction sets and addressing modes) the principles of this work remain valid.

As arithmetic coding is suitable for different order models this offers the possibility of further work for even more efficient object code compression. New techniques are sought which can cope with increasing requirements (such as faster CPU cycle times) without creating insuperable problems to the dense architecture.

Bibliography

[Abrahamson 1989] D M Abrahamson. An adaptive dependency source model for data compression. *Communications of the ACM*, **32**(1):77–83, 1989.

[Abramson 1963] N Abramson. *Information Theory and Coding.* McGraw–Hill, 1963.

[Alexander & Wortman 1975] W G Alexander and D B Wortman. Static and dynamic characteristics of XPL programs. *Computer*, pages 41–46, November 1975.

[Bassiouni *et al.* 1988] M A Bassiouni, N Ranganathan, and A Mukherjee. Software and hardware enhancement of arithmetic coding. In *Lecture Notes in Computer Science*, Rome Italy, June 1988. Fourth International Working Conference SSDBM.

[Bell 1987] T C Bell. *A Unifying Theory and Improvement for Existing Approaches to Text Compression.* PhD thesis, Department of Computer Science, University of Canterbury, New Zealand, 1987.

[Bell *et al.* 1989] T C Bell, I H Witten, and J G Cleary. Modeling for text compression. *ACM Computing Surveys*, **21**(4), December 1989.

[Bell *et al.* 1990] T C Bell, J G Cleary, and I H Witten. *Text Compression.* Prentice Hall, Englewood Cliffs NJ, 1990.

[Bennett & Smith 1989] J P Bennett and G C Smith. The need for reduced byte stream instruction sets. *The Computer Journal*, **32**(4):370–373, 1989.

[Bennett 1988] J P Bennett. *A Methodology for Automated Design of Computer Instruction Sets.* PhD thesis, University of Cambridge, March 1988.

[Cleary & Witten 1984a] J G Cleary and Witten. A comparison of enumerative and adaptive codes. *IEEE Transaction on Information Theory*, **IT-30**(2), March 1984.

[Cleary & Witten 1984b] J G Cleary and I H Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, **COM-32**(4):396–402, April 1984.

[Cmelik *et al.* 1991] R F Cmelik, S I Kong, D R Ditzel, and E J Kelly. An analysis of MIPS and SPARC instruction set utilization on the SPEC benchmarks. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Language and Operating Systems*, pages 290–302, April 1991.

[Cockcroft 1991] A Cockcroft. Sun performance tuning overview. Sun User 1991, September 1991.

[Colwell *et al.* 1985] R Colwell, C Hitchcock, E Jensen, H Brinkley-Sprunt, and C Kollar. Computers, complexity, and controversy. *Computer*, September 1985.

[Colwell *et al.* 1987] R P Colwell, R P Nix, J J O'Donnell, D B Papworth, and P K Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987.

[Cook & Lee 1982] R P Cook and I Lee. A contextual analysis of Pascal programs. *Software Practice and Experience*, pages 195–203, February 1982.

[Cook 1990] R P Cook. An empirical analysis of the Lilith instruction set. *IEEE Transaction on Computers*, **38**(1), January 1990.

[Cornack & Horspool 1987] G V Cornack and R N Horspool. Data compression using dynamic markov modeling. *Computer Journal*, **30**:541–550, December 1987.

[DePrycker 1982] M DePrycker. On the development of a measurement system for high level language program statistics. *IEEE Transaction on Computers*, pages 883–891, September 1982.

[Duncan 1990] R Duncan. A survey of parallel computer architectures. *IEEE Computer*, pages 5–16, February 1990.

[Eickemeyer & Patel 1988] R J Eickemeyer and J H Patel. Performance evaluation of on-chip register and cache organizations. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 64–72, 1988.

[Farrens & Park 1991] M K Farrens and A Park. Dynamic base register caching: A technique for reducing address bus width. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 150–159, May 1991.

[Farrens & Pleszkun 1989] M K Farrens and A R Pleszkun. Improving performance of small on-chip instruction caches. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 234–241, 1989.

[Farrens & Pleszkun 1991] M K Farrens and A R Pleszkun. Strategies for achieving improved processor throughput. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 362–369, May 1991.

[Fiala & Greene 1989] E R Fiala and D H Greene. Data compression with finite windows. *Communications of the ACM*, pages 490–505, April 1989.

[Fisher 1987] J A Fisher. VLIW architectures: Supercomputing via overlapped execution. In *Proceedings of the Second International Conference on Supercomputing*, May 1987.

[Flynn & Hoevel 1984] M J Flynn and L W Hoevel. Measures of ideal execution architectures. *IBM Journal of Research and Development*, **28**(4):356–369, July 1984.

[Foster & Gonter 1971] C C Foster and R Gonter. Conditional interpretation of operation codes. *IEEE Transaction on Computers*, pages 108–111, January 1971.

[Fox *et al.* 1986] E R Fox, K J Kiefer, R F Vangan, and S P Whalen. Reduced intruction set architecture for a GaAs microprocessor system. *IEEE Computer*, pages 71–81, October 1986.

[Fritsch *et al.* 1990] C Fritsch, T Sanchez, and J Anaya. Primitive based architectures. *Computer Architecture News*, 1990.

[Gallager 1978] R G Gallager. Variations on a theme by huffman. *IEEE Transactions on Information Theory*, **IT-24**:668–674, November 1978.

[Gänsheimer & Reisch 1991] W Gänsheimer and J Reisch. Mehr MIPS mit Mips. *c't Magazin für Computer Technik*, pages 228–243, September 1991.

[Glass 1991] B Glass. SPARC revealed. *Byte*, pages 295–302, April 1991.

[Gonzalez-Smith & Storer 1985] M Gonzalez-Smith and J Storer. Parallel algoritms for data compression. *Journal of the ACM*, **32**(2):344–373, April 1985.

[Grohoski 1990] G F Grohoski. Machine organisation of the IBM RISC System/6000 processor. *IBM Journal on Research and Development*, **34**(1):37–58, January 1990.

[Gupta *et al.* 1991] A Gupta, J Hennessy, K Gharachorloo, T Mowry, and W D Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 254–263, 1991.

[Hammerstrom & Davidson 1977] D W Hammerstrom and E S Davidson. Information content of CPU memory referencing behaviour. In *Proceedings of the 7th Annual Symposium on Computer Architecture*, pages 184–192, March 1977.

[Hawthorn 1982] P Hawthorn. Microprocessor assisted tuple access decompression and assembly for statistical database systems. In *Proceedings VLDB*, pages 223–233, 1982.

[Held 1987] G Held. *Data Compression.* John Wiley and Sons, 1987.

[Hennessy & Jouppi 1991] J L Hennessy and N P Jouppi. Computer technology and architectures: An evolving interaction. *IEEE Computer*, pages 18–29, September 1991.

[Hennessy & Patterson 1990] J L Hennessy and D A Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 1990.

[Hopkins 1987] M E Hopkins. A perspective on the 801/reduced instruction set computer. *IBM Systems Journal*, **26**(1), 1987.

[Huffman 1952] D Huffman. A method for the construction of minimum redundancy codes. In *Proceedings IRE 40*, pages 1098–1101, 1952.

[Hwu & Chang 1988] W W Hwu and P P Chang. Exploiting parallel microprocessor microarchitectures with a compiler code generator. In *Proceedings of the International Symposium on Computer Architecture*, June 1988.

[Kaeli & Emma 1991] D R Kaeli and P G Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, May 1991.

[Katevenis 1984] M G H Katevenis. *Reduced Instruction Set Computer Architecture for VLSI.* The MIT Press, 1984. ACM Doctoral Dissertation Award.

[Klaiber & Levy 1991] A C Klaiber and H M Levy. An architecture for software-controlled data prefetching. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 43–53, May 1991.

[Knuth 1971] D E Knuth. An empirical study of Fortran programs. *Software Practice and Experience*, pages 105–133, 1971.

[Krick & Dollas 1991] R F Krick and A Dollas. The evolution of instruction sequencing. *IEEE Computer*, pages 5–15, April 1991.

[Kurian *et al.* 1991] L Kurian, P T Hulina, L D Coraor, and D N Mannai. Classification and performance evaluation of instruction buffering techniques. In *Proceedings* *of the 18th Annual International Symposium on Computer Architecture*, pages 150–159, May 1991.

[Langdon 1984] G G Langdon. An introduction to arithmetic coding. *IBM Journal on Research and Development*, **28**(2), March 1984.

[Lea 1978] R Lea. Text compression with associative parallel processors. *The Computer Journal*, **21**(1):45–56, 1978.

[Mitchell & Pennebaker 1988] J L Mitchell and W B Pennebaker. Optimal hardware and software arithmetic coding procedures for the Q-coder. *IBM Journal on Research and Develpment*, **32**(6):727, November 1988.

[Moffat 1990] A Moffat. Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, **38**(11), November 1990.

[Mukherjee & Bassiouni 1987] A Mukherjee and M Bassiouni. A VLSI chip for efficient transmission and retrieval of information. In *Proceedings 10th ACM SIGIR International Conference on Resaerch and Development in Information Retrieval*, pages 208–216, June 1987.

[Ousterhout 1990] J K Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the USENIX Summer Conference*, pages 247–256, June 1990.

[Patterson & Séquin 1982] D A Patterson and C H Séquin. A VLSI RISC. *Computer*, pages 8–21, September 1982.

[Schoepke & Smith 1993] O S Schoepke and G C Smith. A fast decoding algorithm for arithmetic coding. *Data Compression Conference, Snowbird*, 1993. to be published.

[Schoepke 1992a] O S Schoepke. Compressed code execution. In *Proceedings of the Navy Data Compression Conference, Snowbird*, March 1992.

[Schoepke 1992b] O S Schoepke. A dense instruction set computer architecture proposal. In *XVIII Conference Latinoamericana de Informatica, Las Palmas*, August 1992.

[Schoepke 1992c] O S Schoepke. Entropy in the SPARC instruction set. In *ICCI Proceedings of the International Conference on Computing and Information, Toronto*, pages 5–8, May 1992.

[Schoepke 1992d] O S Schoepke. Executing compressed code: A new approach. In *Proceedings of the Data Compression Conference, Snowbird*, page 396, March 1992.

[Schoepke 1992e] O S Schoepke. A high level VLSI design for ultra dense instruction set computer architectures. In *Proceedings of the Canadian Conference on VLSI, Halifax*, pages 1–8, October 1992.

[Schoepke 1992f] O S Schoepke. Reduce bus traffic through compact encoding. In *International Conference on Parallel Processing, St. Charles*, August 1992. unpublished.

[Schoepke 1992g] O S Schoepke. Ultra dense instruction set computer architecture. In *Proceedings of the International Symposium on Computer and Information Sciences VII, Antalya*, pages 81–87, November 1992.

[Schoepke 1992h] O S Schoepke. Using the entropy in the SPARC instruction set. In *IEEE Proceedings of the International Conference on Computing and Information, Toronto*, pages 6–9, May 1992.

[Shannon 1948] C E Shannon. A mathematical theory of communication. *Bell System Technical Journal*, **27**:379–423 and 623–656, 1948.

[Stallman 1989] R M Stallman. Using and porting GNU cc. Technical report, Free Software Foundation, September 1989.

[Stone 1990] H S Stone. *High-Performance Computer Architecture*. Addison-Wesley, 1990.

[Storer & Szymanki 1982] J A Storer and T G Szymanki. Data compression via textual substitution. *Journal of the ACM*, **29**(4):928–951, October 1982.

[Sun Microsystems Inc. 1987] Sun Microsystems Inc. *Sun SPARC Architecture Manual*, Revision A edition, October 1987.

[Sun 1990] Sun. Sun systems and their caches. Sales Tactical Engineering, June 1990.

[Sweet & Sandman 1982] R E Sweet and J G Jr. Sandman. Empirical analysis of the Mesa instruction set. In *Proceedings of the ACM*, pages 235–243, March 1982.

[Tanenbaum 1978] A S Tanenbaum. Implications of structured programming for machine architecture. *Communications of the ACM*, **21**(3), March 1978.

[Thomas 1991] K Thomas. Entropy. *Dr. Dobb's Journal*, **16**(2):32–34, February 1991.

[Thorlin 1967] J F Thorlin. Code generation for PIE(parallel instruction execution) computers. In *Spring Joint Computer Conference*, April 1967.

[Wade & Stigall 1975] J F Wade and P D Stigall. Instruction design to minimize program size. In *Proceedings 2nd Annual Symposium on Computer Architecture*, pages 41–44, 1975.

[Wang & Wu 1991] L Wang and C Wu. Distributed instruction set computer architecture. *IEEE Transactions on Computers*, **40**(8):915–933, August 1991.

[Weiker 1984] R P Weiker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, **27**(10), October 1984.

[Welch 1984] T A Welch. A technique for high-performance data compression. *Computer*, pages 8–19, June 1984.

[Witten *et al.* 1987] I H Witten, R M Neal, and J G Cleary. Arithmetic coding for data compression. *Communications of the ACM*, **30**(6), June 1987.

[Ziv & Lempel 1977] J Ziv and A Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, **IT-23**(3), May 1977.

[Ziv & Lempel 1978] J Ziv and A Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, **IT-24**(5), September 1978.