A Dense Instruction Set Computer Architecture Proposal

Olaf S. Schoepke
School of Mathematical Sciences
University of Bath
Bath, Avon BA2 7AY, United Kingdom

Abstract

In this paper we propose a dense instruction set architecture which offers some advantages over existing architectures. We approach the problem from different points of view: Entropy measurements show high redundancy in instruction streams, especially when treated as higher order Markov sources; the memory fetch time for instructions has been a major bottleneck for many years, especially for multiprocessors with shared memory systems; ever larger programs are in use which require more and more memory; networks often wasting their time transfering redundant information, and loading times, and bus traffic increase with larger programs. We suggest a computer architecture which can eliminate the problems mentioned above using dense instruction sets.

Keywords: Dense Instruction Set, Compact Code, Entropy

1 Introduction

This paper proposes a new computer architecture based on dense computer instruction sets built for RISC processors. Dense instruction set architectures are important primarily for three reasons: 1) Memory fetch time for instructions has been a bottleneck for many years, due to ever faster processors and multiprocessor systems with shared memory. 2) Entropy measurements show high redundancy in instruction streams, especially when treated as higher order Markov sources. 3) Program size is still increasing, resulting in longer loading times, more paging and swapping activities, and more memory–processor bus traffic. The memory–processor performance gap is clearly a problem on the horizon, and simply making caches larger cannot eliminate this problem. Therefore we introduce a dense instruction set computer architecture which offers some advantages over existing architectures due to much smaller program size. Arithmetic coding, a technique which can get as close as desired to the entropy, is used to encode and decode the instruction stream. Encoding has to be done on static code prior to execution, and decoding on dynamic code during execution.

2 Entropy of Instruction Sets

This section shows the program entropies gathered from observations of the SPARC[1] [10] architecture. Sample programs from which this information is collected are two C compilers, a debugger, part of a window system, part of the operating system and a statistics program. All these statistics are program dependent, because different classes of applications typically use different language features [11]. However, statistics about static characteristics of programs are based on more than 300,000 instructions, dynamic characteristics on millions of instructions.

Static characteristics of programs are interesting for code compaction, program loading times, swapping and paging, and transfer times over the network. Program size becomes more and more important since memory cycle time has not decreased with the same speed as the processor cycle time has increased, and the gap is still increasing. One of the first such studies of programming languages was published by Knuth in 1971 [4], but there are several other publications about static program behaviour.

Dynamic characteristics of programs [3, 8, 11] are of help in the area of program speed and the overall program run time. They are generally more interesting than static characteristics, but they are also more difficult to collect.

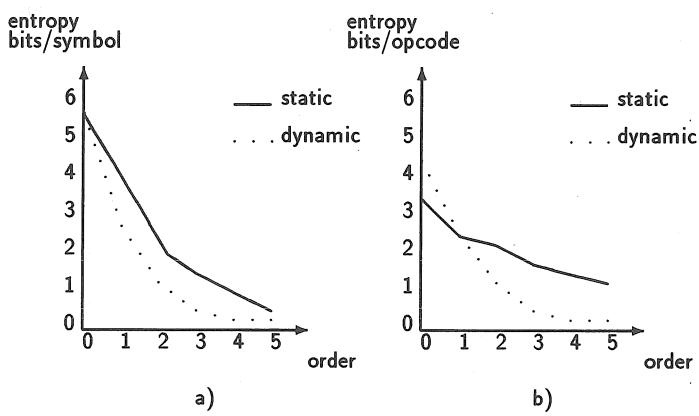

Figure 1: Average entropy of m-th order Markov source in bits/symbol and bits/opcode.

Figure 1 shows the calculated entropy for a m-th order Markov source [1] for the static as well as the dynamic case built with eight bit long symbols in case a) and opcodes [10] in case b) from the instruction stream. For eight bit long symbols as well as opcodes

[1]SPARC is a trademark of Sun Microsystems, Inc.

entropy decreases significantly for the fifth order Markov source from the zero order Markov source. These results show that 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 low entropies as shown in Figure 1 imply we could generate more compact code [9] because the entropy $H(S)$ gives the number of bits for optimal encoding. It is not surprising that we reach a lower entropy in the dynamic case than in the static case, because equal instruction sequences appear more often, e.g. in loops or frequently called procedures.

3 Arithmetic Coding

Recently, arithmetic coding [6, 13] has aroused increased interest as a technique that gives high compression efficiency for a variety of data types as well as being amenable to efficient software and hardware implementations [2]. Arithmetic coding is a coding technique which can get as close as desired to the entropy. It is easy to see that symbols with high probabilities of occurrence and hence larger intervals have less effect on narrowing the given interval than symbols with smaller probabilities.

Arithmetic coding is a data compression technique that encodes data by creating a code string which represents a fractional number 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 is growing. Successive symbols of the message reduce the size of the interval in accordance with the symbol probabilities generated by the given model. The more likely symbols reduce the range by less than the unlikely symbols and hence add fewer bits to the message.

Due to the complexity of most compression methods, past implementations of data compression techniques have mostly been restricted to software. A hardware design of the LZW algorithm using hash tables is briefly discussed in [12] and a fast VLSI implementation of the Huffman's scheme is given in [7]. The arithmetic coding scheme consists of arithmetic operations like addition and multiplication which can be implemented in hardware. Bassiouni [2] describes a high level VLSI chip for the implementation of arithmetic coding.

4 Suggested 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 give a brief overview on how the compiling has to be done to detect branch targets during the encoding process

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

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

Figure 2: Encoding Algorithms

and how the decoder can examine the dense instruction stream to provide instructions to the execution unit for correct execution.

As we can see in the encoding algorithms provided in Figure 2 a new phase of encoding has to be started at each branch target. This of course influences the compression results especially as we know that only a few number of instructions are between branch instruction and branch target.

The decoding algorithms (Figure 3) show us a brief summary of the decoding and execution phase. Building an instruction depends on the number of instruction streams, the size of the symbol to be decoded and the type of the instructions to be built. For example building a 32 bit instruction needs 4 symbols each 8 bit long.

5 Instruction Execution

In this section we take a closer look at how instructions are executed. First we examine the common phases for von Neumann architectures to execute an instruction, and second, we show how to execute dense object code.

Common Instruction Execution

A program for a von Neumann machine consists of a set of instructions that are

```
Decoding_Program ()
{
  Initialize_Arithmetic_Decoder ();
  read ( symbol );
  while !( EOF )
  {
     Decode_Symbol ( symbol );
     Build_Instruction ( symbol );
     if ( instruction == branch_target )
       Initialize_Decoder ();
     else
       if ( instruction == branch_instruction )
         Decoding_Procedure ( instruction );
       else
         Execute_Instruction ( instruction );
    read ( symbol );
  }
}

Decoding_Procedure ( instruction )
{
  Execute_Branch_Instruction ();
  if !( branch_taken )
    return;
  else
  {
    Initialize_Arithmetic_Decoder ();
    Determine_New_Symbol_Address ();
    return;
  }
}
```

Figure 3: Decoding Algorithms

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 vast majority of present–day computers are von Neumann machines. The three distinct phases that constitute the sequencing of each instruction for these architectures are: 1) Determining the memory address that contains the instruction. 2) Fetching the instruction from memory and 3) Executing the instruction.

Before a processor can execute an instruction it must fetch the instruction from memory. Before this operation, the instruction pointer has to 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

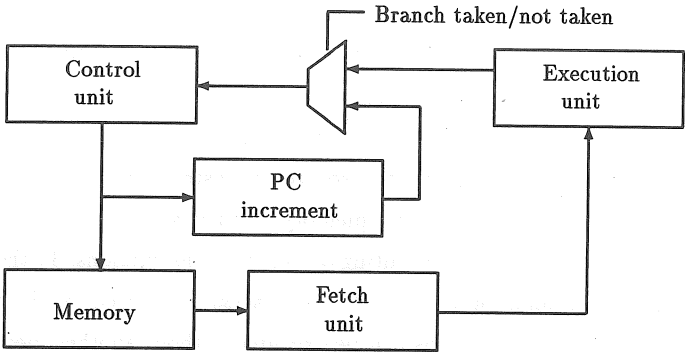

Figure 4: Instruction Execution

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. Therefore, the memory does not operate independently of the processor. Figure 4 shows an example of a von Neumann architecture.

Executing Dense Instructions

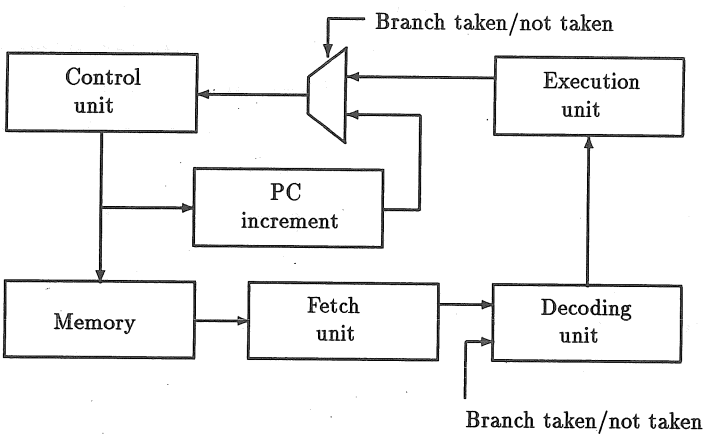

Figure 5: Executing Dense Instruction Sets

Executing dense instruction sets is more complicated than executing conventional object code. Operating independently of the execution unit the fetch unit fetches the instruction from memory. The decoding unit decodes the instruction and stores them for the execution unit to read. This could also be done in a pipeline structure to achieve

better performance.

Before executing an instruction new information has to be fetched if required from the main memory through the fetch unit into the decoder. In the decoding unit, the byte stream has to be decoded and transformed into an executable instruction. Having done that, the execution unit is able to execute the instruction. Because of the dense instruction stream provided by the main memory, it is not always necessary to fetch new information for the decoder after executing one or more instructions. In the worst case, e.g. after taking conditional branches, new information is required for the decoder, but in the best case, several instructions can be executed without another memory access.

Figure 5 gives us an example of a suggested architecture to execute dense object code with only one instruction stream to be decoded. For decoding several streams reflecting the structure given in an instruction stream, the given architecture has to be multiplied by the number of streams to be decoded including an on chip connection between the decoding units for communication. Fetch and decoding units can operate independently of the execution unit. Therefore, the decoding unit can decode the next instruction in advance like in any other pipeline structure.

One of the most difficult problems is how to handle branches, because branch instructions alter the control flow of the program from sequential execution. Like sequencing instructions [5] and cache memories, dense object code execution suffers after having taken branches, because the new memory address has to be calculated and new information is necessary to continue decoding the instruction stream. Furthermore we lose some structure given in the instruction stream because we cannot rely on information further back. The decoding algorithm has to be restarted with only little knowledge of the context and at an addressable memory location.

6 Conclusion

Because the memory–processor performance gap is clearly a problem in the near future, architectures are needed which can speed up the supply of instructions to the execution. We suggest a new architecture based on dense instruction sets which offers some advantages over existing architectures. On the one hand, the instruction stream has to be decoded before the instruction can be executed by the execution unit. On the other hand, the time used to transfer data especially between several processors and shared memory will drop sharply. Smaller program size also means less swapping, shorter loading times, and minor traffic on a network. With the method presented, it is possible to reduce program size considerably and thus to narrow the increasing gap

between memory cycle time and processor cycle time.

References

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

[2] MA Bassiouni, N Ranganathan, A Mukherjee *Software and Hardware Enhancement of Arithmetic Coding* Lecture Notes in Computer Science, Fourth International Working Conference SSDBM, Rome, Italy, June 1988

[3] JL Hennessy, DA Patterson *Computer Architecture A Quantitative Approach* Morgan Kaufmann Publishers Inc., 1990

[4] DE Knuth *An Empirical Study of FORTRAN Programs* Software Practice and Experience, pp. 105–133, 1971

[5] RF Krick, A Dollas *The Evolution of Instruction Sequencing* IEEE Computer, pp. 5–15, April 1991

[6] GG Langdon *An Intr. to Arithmetic Coding* IBM J. Res. Dev., vol 28 no 2, 1984

[7] A Mukherjee, M Bassiouni *A VLSI chip for efficient transmission and retrieval of information* Proc. 10th ACM SIGIR International Conf. on Research and Development in Information Retrieval, pp. 208–216, June 1987

[8] DA Patterson, CH Sequin *A VLSI RISC* Computer, pp. 8–21, September 1982

[9] OS Schoepke *Using the Entropy in the SPARC Instruction Set* IEEE Proc. of the International Conference on Computing and Information, May 1992

[10] Sun SPARC Architecture Manual Revision A, October 1987

[11] RP Weiker *Dhrystone: A Synthetic Systems Programming Benchmark* Communications of the ACM, vol 27 no 10, October 1984

[12] TA Welch *A Technique for High–Performance Data Compression* Computer, pp. 8–19, June 1984

[13] IH Witten, RM Neal, JG Cleary *Arithmetic Coding for Data Compression* Communications of the ACM, vol 30 no 6, June 1987