The Effect of Instruction Set Complexity on Program Size and Memory Performance

Jack W. Davidson

Richard A. Vaughan

Computer Science Technical Report TR-87-12
June 18, 1987

The Effect of Instruction Set Complexity on Program Size and Memory Performance*

JACK W. DAVIDSON
RICHARD A. VAUGHAN

University of Virginia

One potential disadvantage of a machine with a reduced instruction set is that object programs may be substantially larger than the object programs for a machine with a richer, more complex instruction set. The main reason is that a small instruction set will require more instructions to implement the same function. In addition, the tendency of RISC machines to use fixed length instructions with a few instruction formats contributes to increased object program size. It has been conjectured that the resulting larger programs could adversely affect memory performance and bus traffic. In this paper we report the results of a set of experiments to isolate and determine the effect of instruction set complexity on cache memory performance and bus traffic. Three high-level language compilers were constructed for machines with instruction sets of varying degrees of complexity. Using a set of benchmark programs, we evaluated the effect of instruction complexity had on program size. Five of the programs were selected and address traces of their execution were obtained for each machine. These traces were used to perform a set of trace-driven simulations to study each machine's cache and bus performance. We found that the miss ratio was severely affected by the increase in program size due to the simplicity of the instruction set. Our measurements of bus traffic show that even with relatively large cache memories, machines with simple instruction sets can expect two to three times the bus traffic of machines with complex instruction sets. While it appears that the degradation in miss ratio can be corrected simply by increasing the size of the cache, reducing the amount of bus traffic due to increased program size may be problematic.

1. INTRODUCTION

One of the primary goals of a computer architect is the design and construction of machines that support the efficient execution of the programs that will run on them. A number of new architecture design principles have evolved for guiding the design of machines to support the execution of high-level languages. The distinguishing characteristic of the machines constructed using these principles is the reduced number of operations contained in the instruction set. Consequently, these machines have been termed RISCs—*reduced instruction set computers* [12]. Patterson [14] lists some of the RISC design principles:

1. Functions should be kept simple unless there is a very good reason to do otherwise.
2. Microinstructions should not be faster than simple instructions.
3. Moving software into microcode does not make it better.

*This work was supported in part by the National Science Foundation under Grant CCR-8611653

4. Simple decoding and pipelined execution are more important than program size.
5. Compiler technology should be used to simplify instructions rather than to generate complex instructions.

Both experimental and commercial machines have been built using these principles. These machines have several characteristics in common. Patterson [14] and Hennessy [8] identify these characteristics.

1. Operations are register-to-register, with only LOAD and STORE operations accessing memory.
2. The operations and addressing modes are reduced.
3. A fixed length instruction and a small variety of instruction formats.
4. RISC branches avoid pipeline penalties.

The simplicity of the instruction set provides for a number of implementation advantages that can substantially enhance the performance of the machine. For example, the restriction that arithmetic and logical operations be register-to-register may permit the number of pipeline stages and/or their duration to be reduced resulting in faster execution. The reduced number of operations and addressing modes may make it possible to produce a hardwired control unit. If the implementation is microprogrammed, silicon resources may be freed for the implementation of other features to enhance performance. Possibilities included, on-chip instruction and data caches, instruction buffers, and larger register sets. Finally, the use of fixed length instructions and a few formats permits simpler, faster instruction decoding.

A disadvantage of a RISC machine is that the memory requirements of programs may be substantially greater than those of programs on a complex instruction set machine (CISC). The number of instructions to implement a given function on a RISC machine will normally be much greater than the number required for a CISC machine. In addition, CISC machines usually have densely encoded instructions and a variety of instruction formats that further reduces the size of object programs. It has been conjectured that the larger size of object programs on RISC machines may negatively affect memory performance [20].

Normally, comparisons between RISC machines and CISC machines are difficult because the architectures differ in more ways than just the complexity of their instruction sets. To perform fair experiments and so that no unseen architectural bias is introduced, we used a technique called architectural subsetting [14]. Three virtual machines with instruction sets of varying degrees of complexity were created and a high-level language compiler was constructed for each machine. The compilers were constructed using a state-of-the-art automatic code generation system. For each machine, a set of benchmark programs were compiled and the size of the resulting programs were compared across machines. Five of these programs were then selected, and for each of the three machines, an address trace of their execution was obtained. These address traces were used to perform trace-driven simulations of cache memories of various sizes. We used the trace-driven simulations to evaluate the effect instruction set complexity had on cache memory performance and bus traffic.

This paper has the following organization. Section 2 describes the three instruction sets used in the experiments. The programming language used and how the compilers were constructed are described in Section 3. The effects of instruction set complexity on object program size and on memory memory performance are reported in Section 4. Conclusions are presented in Section 5.

2. THREE INSTRUCTION SETS

Because of differences in implementation and software, designing and conducting experiments that fairly and without bias compare architectures is difficult. For example, measuring the execution speed of a high-level language program on different architectures may not reveal anything about the quality of the respective architectures. Differences in implementation, packaging, and software all affect the outcome of the measurements. This is not to say that such comparisons are not useful. It is, after all, the speed of the system (hardware and software) that counts. However, such measurements are often used erroneously as indicators of architectural quality.

To perform a set of experiments where such sources of bias are eliminated as much as possible, we used a technique called architectural subsetting [14]. In this technique, the instruction set of a virtual machine is created by selectively choosing instructions and addressing modes from an existing machine. The resulting virtual machine is a subset of the base architecture. For the experiments reported here, we created three subsets of the VAX architecture. The richness of the VAX instruction set makes it ideal for architectural subsetting. The virtual machines represented by these subsets had instructions sets of varying degrees of complexity. The three machines are called MAXVAX, MIDVAX, and MINVAX. Because the machines are subsets of the same architecture and thus have the same underlying implementation, differences in instruction formats and operand and addressing mode encodings are eliminated. The major features of these machines are described in the following sections. Appendix A contains a complete description of the instruction sets of all three machines.

2.1 MAXVAX

The MAXVAX instruction set is the part of the full VAX instruction set necessary to support the compilation of Y programs. Y [7] is a high-level programming language that is similar in many respects to C. Section 3 contains a brief introduction to Y. The MAXVAX includes 16 addressing modes and most instructions can use any addressing mode as both a source and destination. Both two- and and three-address forms of the arithmetic and logical operations are included. In addition, the instruction set contains special forms of common operations. For example, the instruction set includes increment and decrement operations, a clear operation, and special instructions for pushing operands and addresses on the runtime stack. It also contains instructions that are designed for efficient implementation of switch statements and loops.

2.2 MIDVAX

In contrast to the MAXVAX instruction set, the MIDVAX instruction set contains only eight addressing modes. The instruction set is reduced further by allowing only two-address forms of arithmetic and logical instructions. While the source operand can be any of the MIDVAX addressing modes, the destination operand must be a register. All special case instructions such as increment, decrement, clear, as well as the special stack instructions and loop instructions are omitted.

2.3 MINVAX

The MINVAX instruction set was constructed so as to approximate the instruction set of a typical RISC machine. Consequently, the MINVAX instruction set contains only four simple addressing modes. All arithmetic and logical operations are two-address instructions and are register-to-register. Memory is accessed through load and store type instructions only.

3. THREE COMPILERS

Three compilers for the Y programming language for the MAXVAX, MIDVAX, and MINVAX machines were constructed. Y [7] is a structured, general-purpose programming language intended for use in systems programming applications, such as those described in *Software Tools* [10]. Syntactically and semantically, Y is similar to C. It supports separate compilation, recursive procedures, structured control flow constructs, and expressions involving scalars, and arrays of integers, characters, and reals. From a code generation point of view, Y is equivalent to C or Pascal.

The code generators for the Y compilers were constructed using PO, a retargetable peephole optimizer [4]. PO is particularly well-suited to architectural experimentation for several reasons. It is quickly and easily retargeted to a new machine by supplying a description of the target machine's instruction set, and it is able to exploit any of the target machine's special operations and addressing modes. Finally, PO performs a number of optimizations that one would expect a production compiler to perform. We point out that because the code selector of the compiler is constructed automatically by supplying a new machine description, the quality of the code emitted by the compiler has less dependence on the skill of the implementors than other code generation techniques. A schematic of a retargetable compiler that uses PO is shown in Figure 1.

PO is made up of three distinct phases called Cacher, Combiner, and Assigner. Each phase operates on register transfer lists ('RTLs') which describe an instruction's effect. For example, a PDP-11 instruction that adds two registers would be expressed in the RTL notation as:

```
        r[2] = r[2] + r[3]; CC = r[2] + r[3] ? 0;
```

Any RTL is machine specific, but the form of the RTL is machine independent.

While PO can be used as a general-purpose peephole optimizer, it can also be used to replace a conventional code generator found in a traditional compiler. Compilers developed using PO in this way use two pervasive strategies that lead to the generation of excellent code. The first strategy is to have the front-end generates naive, but correct code for an abstract machine. The code expander translates the abstract machine code into straightforward code for the target machine. The code is inefficient, but simple to produce. Both of these phases are concerned only with producing semantically correct code. Efficiency is *not* an issue. The second strategy is to perform all optimizations on object code. The guiding principle is more complete and thorough optimization is possible by operating on object code where knowledge of the target machine can be used. The above two strategies are similar to the approach taken in the PL.8 compiler for the IBM 801 [1,15].

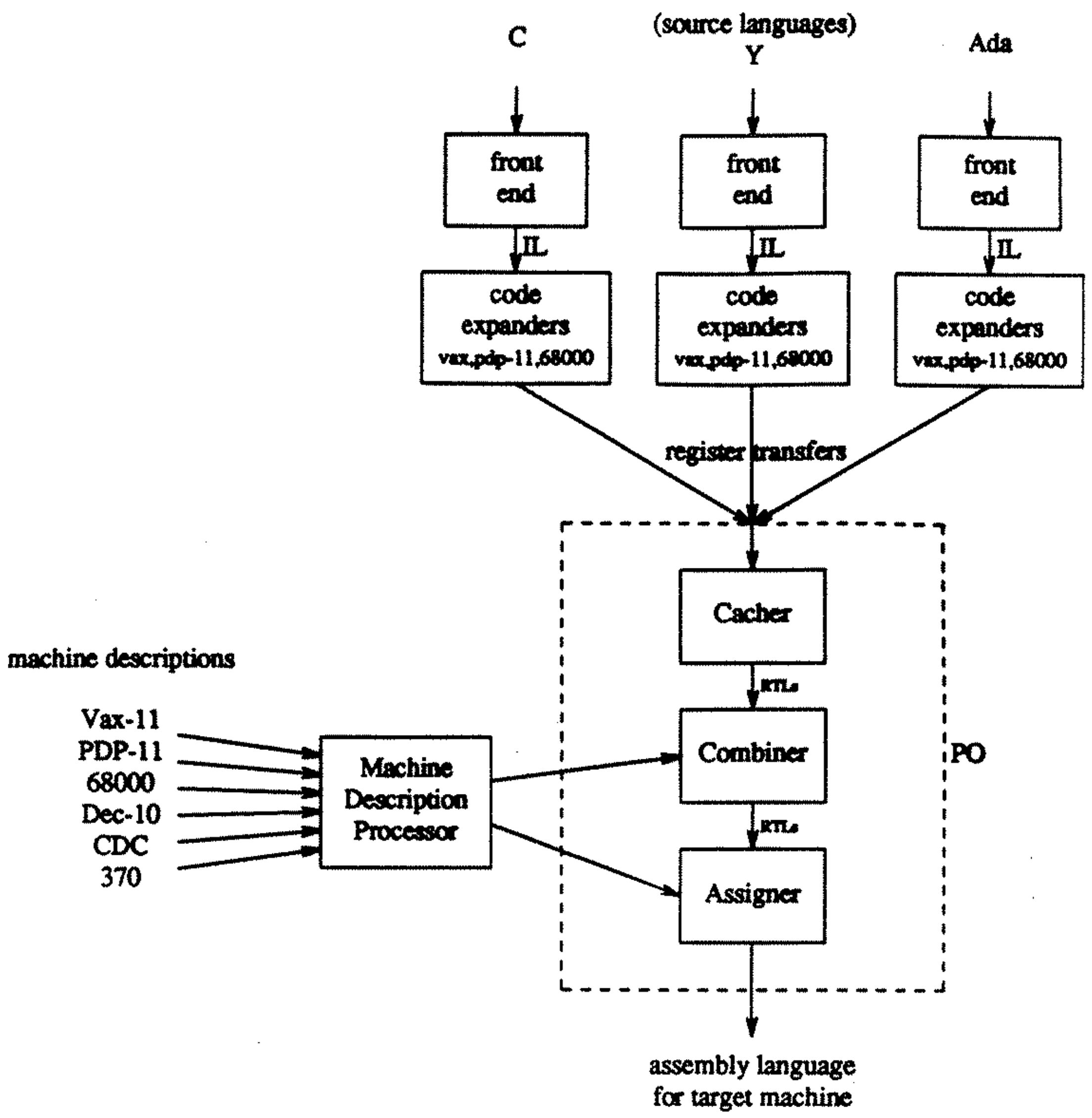
C
(source languages)
Y
Ada
front end
front end
front end
IL
IL
IL
code expanders vax,pdp-11,68000
code expanders vax,pdp-11,68000
code expanders vax,pdp-11,68000
register transfers
Cacher
RTLs
Combiner
RTLs
Assigner
PO
machine descriptions
Vax-11
PDP-11
68000
Dec-10
CDC
370
Machine Description Processor
assembly language for target machine

Figure 1. Schematic of PO Compiler Development System.

Briefly, Cacher performs common subexpression elimination, allocates registers, performs a limited type of flow analysis, and identifies dead variables. Combiner advances over Cacher's output seeking adjacent instructions that can be replaced with singletons. Combiner also performs the following optimizations:

1. targeting [11,17],
2. evaluation order determination [18], and
3. instruction scheduling if the machine includes a pipeline [16].

Once optimization is complete, Assigner does register assignment and translates the RTLs to assembly language. Both Combiner and Assigner are retargeted by supplying a description of the target machine's instruction set. Combiner uses the machine description (MD) to determine whether the combined effect of two instructions can be achieved by a single instruction. The MD is also used by Assigner to convert RTLs to assembly language. Other documents offer a more complete treatment of Cacher and Assigner [3], and PO [4,5].

The code emitted by compilers constructed using PO compares favorably to the code produced by other state-of-the-art code generation systems [4,5]. Indeed, PO has been used to create C compilers for the AT&T 3B5 and the VAX, as well as a validated commercial implementation of Ada.

4. RESULTS

A set of ten Y programs were selected to serve as benchmarks for evaluating the effect of instruction set complexity on object program size. The programs range in complexity from a simple solution to the eight queen's problem to a 3600-line compiler. The ten benchmark programs were:

1. 8q—a recursive solution to the eight queens problem,
2. find—searches for a regular expression in a file,
3. ypp—the Y preprocessor,
4. y—the Y front end,
5. wf—computes the frequency of word use in a file,
6. xd—a text editor,
7. ar—a file archiver,
8. roff—a text formatter,
9. spo—a simple peephole optimizer, and
10. od—a file dump utility.

4.1 Object Code Size

Table I shows the sizes of the object code for the ten programs on the MAXVAX, MIDVAX, and MINVAX machines. The MINVAX machine required an average of 2.5 times more memory than the MAXVAX, while the MIDVAX machine required an average of 1.54 times more memory. We also measured the average instruction size for each machine. For the MAXVAX, MIDVAX, and MINVAX respectively, the average instruction size was 4.10 bytes, 3.71 bytes, and 3.61 bytes. Because MINVAX$\subset$MIDVAX$\subset$MAXVAX, the above numbers indicate the compiler was able to take advantage of the additional instruction set complexity. We note that Emer reported an average instruction size of 3.8 bytes for the complete VAX [6].

Program	Source Lines	MAXVAX		MIDVAX		MINVAX	
		Size	Relative	Size	Relative	Size	Relative
8q	40	1979	1.00	2271	1.14	2979	1.50
find	320	3625	1.00	4665	1.30	5801	1.61
ypp	898	10071	1.00	15931	1.58	25983	2.58
y	3599	37199	1.00	61219	1.65	100203	2.69
wf	169	3075	1.00	3779	1.22	5395	1.75
xd	1289	16349	1.00	23215	1.42	42671	2.61
ar	1403	14321	1.00	20336	1.42	31936	2.23
roff	2134	24025	1.00	40842	1.70	59103	2.46
spo	187	3246	1.00	4707	1.45	8147	2.51
od	723	9736	1.00	13631	1.40	25508	2.62
Average	1076	12363	1.00	19060	1.54	30773	2.48

Table I. Program Size (Bytes) and Ratio to MAXVAX.

There have been other studies that examined the effect of instruction set complexity on program size. In 1981, a group of researchers at the University of California at Berkeley compared the memory requirements of the VAX and RISC I [13]. For the eleven benchmark programs they used, they found that the RISC I programs required an average of 50% more memory than the VAX. In 1985, a group of researchers at Carnegie-Mellon compared the code size of 16 programs on the VAX and RISC II [2]. For those benchmarks, they found that the VAX required on average 3.5 times less memory than RISC II.

The results reported here fall between these two extremes. We believe that the results presented provide a more accurate characterization of the effect instruction set complexity has on program size. Most reports have compared different machines, and consequently do not serve to isolate the effect of instruction set complexity on program size. For example, the Berkeley experiments compared the VAX, a machine with 16 general-purpose registers, to RISC I, a machine with 32 registers. As was noted by the Carnegie-Mellon researchers, CISC machines can also benefit from additional registers.

A second possible source of bias is caused the the code generation technology used to build the compilers for the machines being compared. Many of the previous studies on program size used the Unix portable C compiler (PCC) to generate code for the machines being compared. The code generator for this compiler is retargeted by hand and as Johnson, the author of PCC, notes, retargeting parts of the compiler require hard intellectual effort [9]. The task of getting this compiler to produce "optimal" code for a machine with a large number of operations and many addressing modes is obviously going to be harder than getting it to generate "optimal" code for a machine with a simple instruction set. Consequently, the quality of the emitted code depends, to a large extent, on the ingenuity and perseverence of the person retargeting the compiler. In contrast, the three compilers used for the experiments described here were constructed using an automatic code generation technology. The compilers were retargeted by supplying a description of the target machine's instruction set†. Consequently, differences that could be caused by the skill level of the persons retargeting the compiler are removed. In addition, this code generation system is specifically designed to exploit any special instructions or addressing modes the target machine may have.

The Carnegie-Mellon experiments, on the other hand, compared the size of 16 programs that were handcoded in assembly language. While the current generation of optimizing compilers produce excellent code, for small programs (like the benchmarks used), humans can still probably do a better job of generating code. Consequently, we believe that their measurements of the memory requirements of the VAX are too optimistic.

4.2 Cache Performance

To investigate the effect of program size on cache performance, we used the technique of trace-driven simulation [19]. Five of the benchmark programs were selected and address traces of their execution were obtained for the three machines. The traces were generated by modifying and instrumenting the UNIX debugger, ***adb***, to record every address referenced by the program. The cache simulator measured the *miss ratio* and *bus traffic*. The miss ratio is the number of cache misses divided by the number of cache accesses. The bus traffic is the number of memory requests issued.

The five programs selected were wf, ypp, y, 8q, and xd. When compared to the MAXVAX, these programs required an average of 1.55 times more memory on the MIDVAX, and 2.5 times more memory on the MINVAX. The average size of these programs is similar to the average size of the complete set of benchmark programs. The simulations used all five traces to simulate multiprogramming. Scheduling of which trace to use next was done on a round-robin basis. A context switch was performed every 10,000 time units. A cache hit incremented the clock by one time unit, a cache miss incremented the clock by 10 time units. The traces were well over 250,000 references in length so the effects of cold start could be ignored.

In creating the three compilers a number of steps were taken to ensure that any differences in address traces were due to complexity of the instruction sets and not other factors. The register allocation and assignment phases of the three compilers were given the same number of registers to allocate (eight). In addition, because the

†Because of the strategy of producing naive code in the front end and because because MINVAX$\subset$MIDVAX$\subset$MAXVAX, it was possible to use the same front end for all three compilers.

MINVAX machine is a register-to-register architecture, one would expect it to use more registers than the other two machines. The normal VAX subroutine calling convention requires that a called program save and restore the registers it uses. This could mean that the MINVAX and MIDVAX would be penalized on subroutine calls. To be fair, the three compilers generated code to save and restore on procedure entry and exit all allocable registers regardless of usage. Finally, we instrumented the compilers to report when register spills occurred. The five programs used in these experiments had no register spills.

For the trace-driven simulations, the caches had the following parameters: a) two-way set-associative, b) LRU replacement, c) write-through and no write-allocate, and d) a line size of 16 bytes. The width of the data path between the CPU and the cache is 16 bytes and between the cache and memory is 8 bytes. Throughout the simulations described in this paper, the line size, the number of elements per set, and the replacement algorithm were held constant. Only the cache capacity was varied. In addition, we included the simulation of a simple instruction buffer (IB). The simulated IB issues requests to the cache when there is room for 4 bytes in the buffer. The simulation does not proceed until these bytes are placed in the buffer.

4.2.1 Miss Ratio

Figure 2 contains a plot comparing the miss ratios of the three machines using the address traces of the five programs. As one would expect, the MAXVAX had lower miss ratios than either the MIDVAX or MINVAX. For caches less than 32K bytes, the magnitude of the difference between the MAXVAX and the MINVAX, however, is surprising. For cache capacities of up to 32K bytes, the MINVAX would require a cache that is four times the size of the MAXVAX cache to obtain equivalent or better performance in terms of miss ratio. At cache size of 64k bytes or greater, the machines have essentially the same performance. One of the advantages often cited for a RISC is that the simplified implementation of the control unit may free silicon resources for other uses. Based on the results presented here, these resources may well be needed to compensate for higher miss ratios due to the increased working set size.

4.3 Memory Bus Traffic

An important factor in the design of a computer system is the amount of memory bus traffic. The amount of bus traffic affects the memory bandwidth requirements. Because a machine with a simple instruction set requires more

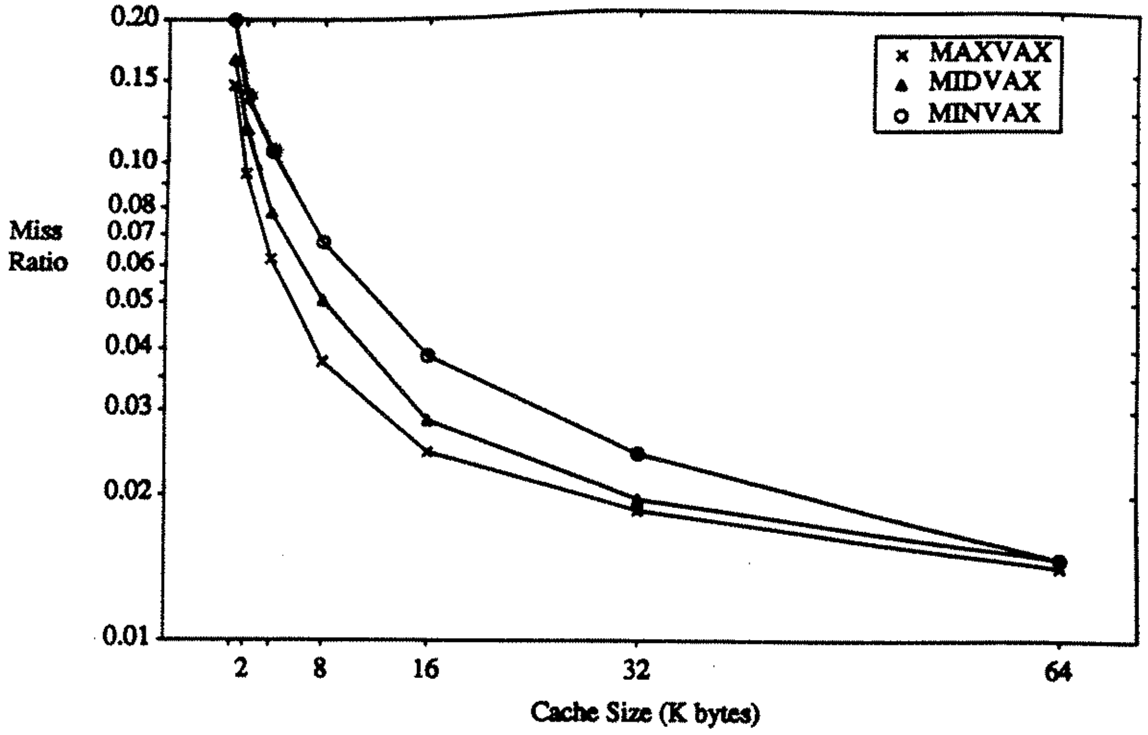

Figure 2. Miss Ratio vs. Cache Size for MAXVAX, MIDVAX, and MINVAX machines.

Cache	MAXVAX		MIDVAX		MINVAX	
Capacity	Traffic	Relative	Traffic	Relative	Traffic	Relative
1	174976	1.00	255860	1.46	471218	2.69
2	116038	1.00	184108	1.58	331350	2.85
4	73354	1.00	119340	1.62	253640	3.45
8	42902	1.00	74848	1.74	159546	3.72
16	25500	1.00	40198	1.57	88770	3.48
32	17750	1.00	25574	1.44	51860	2.92
64	13150	1.00	17024	1.29	28448	2.16
128	12130	1.00	14556	1.20	24381	2.01
256	11467	1.00	13416	1.17	21213	1.85

Table II. Bus Traffic and Ratio to MAXVAX.

instructions to implement a given function, all other things being equal, we can expect such a machine to have higher memory bandwidth requirements than a machine with a complex, tightly encoded instruction set. Table II compares the memory bus traffic for the five benchmark programs running on the MAXVAX, MIDVAX, and MINVAX machines with caches of various sizes. The MINVAX machine has 2.5 to 3.5 times more bus traffic than the MIDVAX machine. Even for a cache size of 256k bytes, the MINVAX machine produces 1.85 times more bus traffic than the MAXVAX machine.

5. DISCUSSION AND SUMMARY

This study has attempted to isolate and evaluate the effect reducing the complexity of an instruction set on program size and memory performance. By using subsets of the same machine to represent machines with instruction sets of varying complexity, we eliminated machine differences that could have affected program size. The machines had the same number of registers, they used the same calling sequence, and they used the same encodings for instructions and addressing modes they had in common. In addition, we used a state-of-the-art code generation system that is capable of exploiting a complex machine's instruction set and addressing modes. We expect that future compilers will be constructed with similar, if not better, code generation technology.

With all other factors constant, we found that simple instruction sets can result in programs that require two and a half times more memory than the same programs on a machine with a complex instruction set. Our evaluation of cache performance showed that for small cache sizes, instruction set complexity can severely affect the miss ratio. Fortunately, this aspect of performance can be corrected through the use of large caches. Finally we examined the amount of bus traffic on the three machines. Even with a large caches (>64k), a machine with a simple instruction set can expect to generate twice as much bus traffic as a machine with a complex instruction set. Overcoming the potential performance bottleneck caused by the increased bus traffic will require innovative high-performance memory systems.

6. REFERENCES

1. Auslander, M. and Hopkins, M., An Overview of the PL.8 Compiler, *Proceedings of the ACM SIGPLAN Notices '82 Symposium on Compiler Construction*, Boston, MA, June 1982, 22-31.
2. Colwell, R. P., III, C. Y. H., Jensen, E. D., Sprunt, H. M. B. and Kollar, C. P., Computers, Complexity, and Controversy, *IEEE Computer 18*, 9 (September 1985), 8-19.
3. Davidson, J. W. and Fraser, C. W., Register Allocation and Exhaustive Peephole Optimization, *Software—Practice and Experience 14*, 9 (September 1984), 857-866.
4. Davidson, J. W. and Fraser, C. W., Code Selection through Object Code Optimization, *Transactions on Programming Languages and Systems 6*, 4 (October 1984), 7-32.
5. Davidson, J. W., A Retargetable Instruction Reorganizer, *Proceedings of the SIGPLAN Notices '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986, 234-241.
6. Emer, J. S. and Clark, D. W., A Characterization of Processor Performance in the VAX-11/780, *Proceedings of the 11th Annual Symposium on Computer Architecture*, Ann Arbor, June 1984, 301-310.
7. Hanson, D. R., The Y Programming Language, *SIGPLAN Notices 16*, 2 (February 1981), 59-68.
8. Hennessy, J. L., VLSI Processor Architecture, *IEEE Transactions on Computers 33*, 12 (December 1984), 1221-1246.
9. Johnson, S. C., A Tour Through the Portable C Compiler, *Unix Programmer's Manual, 7th Edition 2B*, (January 1979), .
10. Kernighan, B. W. and Plauger, P. J., *Software Tools*, Addison-Wesley, Reading, MA, 1976.
11. Leverett, B. W., Cattell, R. G. G., Hobbs, S. O., Newcomer, J. M., Reiner, A. H., Schatz, B. R. and Wulf, W. A., An Overview of the Production Quality Compiler-Compiler Project, CMU-CS-79-105, Carnegie-Mellon University, Pittsburg, PA, February 1979.
12. Patterson, D. A. and Ditzel, D. R., The Case for the Reduced Instruction Set Computer, *Computer Architecture News 8*, 6 (October 1980), 25-33.
13. Patterson, D. A. and Sequin, C. H., RISC I: A Reduced Instruction Set VLSI Computer, *Proceedings of the Eighth Annual Symposium on Computer Architecture*, Minneapolis, MN, May 1981, 443-457.
14. Patterson, D. A., Reduced Instruction Set Computers, *Communications of the ACM 28*, 1 (January 1985), 8-21.
15. Radin, G., The 801 Minicomputer, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, March 1982, 39-47.
16. Rymarczyk, J. W., Coding Guidelines for Pipelined Processors, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, March 1982, 12-19.
17. Schatz, B. R., Algorithms for Optimizing Transformations in a General Purpose Compiler: Propagation and Renaming, RC 6232, IBM Thomas J. Watson Research Center , Yorktown Heights, NY, October 1976.
18. Sethi, R. and Ullman, J. D., The Generation of Optimal Code for Arithmetic Expressions, *Journal of the ACM 17*, 6 (October 1970), 715-728.
19. Smith, A. J., Cache Memories, *Computing Surveys 14*, 3 (September 1982), 473-530.
20. Tanenbaum, A. S., Implications of Structured Programming for Machine Architecture, *Communications of the ACM 21*, 3 (March 1978), 237-246.

7. APPENDIX A

7.1 MAXVAX Instruction Set

Addressing Modes

Mode	RTL Notation	Assembly Language
register	r[N]	rN
register deferred	m[r[N]]	(rN)
direct	m[A]	A
displacement	m[r[N] + C]	C(rN)
autoincrement	m[r[N]++]	(rN)+
autodecrement	m[--r[N]]	-(rN)
autoincrement deferred	m[l[r[N]++]]	*(rN)+
displacement deferred	m[l[r[N] + C]]	*C(rN)
immediate	C	$C
indexed displacement	m[r[X] << SZ + C]	C[rX]
indexed register deferred	m[r[X] << SZ + r[N]]	(rN)[rX]
indexed immediate displacement	m[r[X] << SZ + r[N] + C]	C(rN)[rX]
indexed autoincrement	m[r[X] << SZ + r[N]++]	(rN)+[rX]
indexed autodecrement	m[r[X] << SZ + --r[N]]	-(rN)[rX]
indexed autoincrement deferred	m[r[X] << SZ + m[r[N]++]]	*(rN)+[rX]
indexed deferred displacement	m[r[X] << SZ + m[r[N] + C]]	*C(rN)[rX]

Operations

Operation	RTL Notation	Assembly Language
test	nz = DST ? 0;	tst DST
compare	nz = DST ? SRC;	cmp DST,SRC
bit test	nz = DST & SRC ? 0;	bit SRC,SRC
clear	DST = 0; nz = 0;	clr DST
push long	l[r[14]++] = SRC; nz = SRC ? 0;	pushl DST
move	DST = SRC; nz = SRC ? 0;	mov SRC,DST
convert	DST = cvt(SRC); nz = cvt(SRC) ? 0;	cvt SRC,DST
move address	DST = addr(SRC); nz = addr(SRC) ? 0;	mova SRC,DST
push address	l[r[14]++] = addr(SRC); nz = addr(SRC) ? 0;	pusha SRC
increment	DST = DST + 1; nz = DST + 1 ? 0;	inc DST
decrement	DST = DST - 1; nz = DST - 1 ? 0;	dec DST
addition	DST = SRC1 + SRC2; nz = SRC1 + SRC2 ? 0;	add SRC2,SRC1,DST
subtraction	DST = SRC1 - SRC2; nz = SRC1 - SRC2 ? 0;	sub SRC2,SRC1,DST
multiplication	DST = SRC1 * SRC2; nz = SRC1 * SRC2 ? 0;	mul SRC2,SRC1,DST
division	DST = SRC1 / SRC2; nz = SRC1 / SRC2 ? 0;	div SRC2,SRC1,DST
move negated	DST = -SRC; nz = -SRC ? 0;	mneg SRC,DST
move complemented	DST = ~SRC; nz = ~SRC ? 0;	mcom SRC,DST
bit set	DST = SRC1 | SRC2; nz = SRC1 | SRC2 ? 0;	bis SRC2,SRC1,DST
bit clear	DST = SRC1 & ~SRC2; nz = SRC1 & ~SRC2 ? 0;	bic SRC2,SRC1,DST
shift	DST = SRC1 << SRC2; nz = SRC1 << SRC2;	ashl SRC2,SRC1,DST
exclusive or	DST = SRC1 ^ SRC2; nz = SRC1 ^ SRC2;	xor SRC2,SRC1,DST
jump	pc = LABEL;	jbr LABEL
conditional jumps	pc = nz REL 0 -> LABEL | pc;	jREL LABEL
add compare and branch	pc = DST + SRC <= SRC2 -> LABEL | pc;	acb SRC2,SRC1,DST,LABEL
add one and branch	pc = DST + 1 < SRC -> LABEL | pc;	aoblss SRC,DST,LABEL
add one and branch	pc = DST + 1 <= SRC -> LABEL | pc;	aobleq SRC,DST,LABEL
subtract one and branch	pc = DST - 1 >= SRC -> LABEL | pc;	sobgeq SRC,DST,LABEL
subtract one and branch	pc = DST - 1 > SRC -> LABEL | pc;	sobgtr SRC,DST,LABEL
case	pc = DST - SRC1 < SRC2 -> LABEL | pc;	case DST,SRC1,SRC2
call instruction (stack)	PC = calls(numargs,DST);	calls numargs,DST

Notes: *N* and *X* denote register numbers. *SZ* indicates a shift count that depends of the size of datatype being accessed. *DST* and *SRC* denote any of the addressing modes. The instruction set includes two-address and three-address forms of the arithmetic and logical instructions. *nz* denotes the condition codes; *pc* is the program counter. *REL* denotes any of the six relational operations.

7.2 MIDVAX Instruction Set

Addressing Modes

Mode	RTL Notation	Assembly Language
register	r[N]	rN
immediate	C	$C
register deferred	m[r[N]]	(rN)
autoincrement	m[r[N]++]	(rN)+
autodecrement	m[--r[N]]	-(rN)
direct	m[A]	A
displacement	m[r[N] + C]	C(rN)
displacement deferred	m[l[r[N] + C]]	*C(rN)

Operations

Operation	RTL Notation	Assembly Language
compare	nz = DST ? SRC;	cmp DST,SRC
bit test	nz = DST & SRC ? 0;	bit SRC,SRC
load	r[N] = SRC; nz = SRC ? 0;	mov SRC,rN
store	DST = r[N]; nz = r[N] ? 0;	mov rN,DST
addition	r[N] = r[N] + SRC; nz = r[N] + SRC ? 0;	add SRC,rN
subtraction	r[N] = r[N] - SRC; nz = r[N] - SRC ? 0;	sub SRC,rN
multiplication	r[N] = r[N] * SRC; nz = r[N] * SRC ? 0;	mul SRC,rN
division	r[N] = r[N] / SRC; nz = r[N] / SRC ? 0;	div SRC,rN
negate	r[N] = -r[N]; nz = -r[N] ? 0;	mneg rN,rN
complement	r[N] = ~r[N]; nz = ~r[N] ? 0;	mcom rN,rN
convert	r[N] = cvt(SRC); nz = cvt(SRC) ? 0;	cvt SRC,rN
bit set	r[N] = r[N] | SRC; nz = r[N] | SRC ? 0;	bis SRC,rN
bit clear	r[N] = r[N] & ~SRC; nz = r[N] & ~SRC ? 0;	bic SRC,rN
shift	r[N] = r[N] << SRC; nz = r[N] << SRC;	ashl SRC,rN,rN
exclusive or	r[N] = r[N] ^ SRC; nz = r[N] ^ SRC;	xor SRC,rN,rN
jump	pc = LABEL;	jbr LABEL
conditional jumps	pc = nz REL 0 -> LABEL | pc;	jREL LABEL
call instruction (stack)	PC = calls(numargs,DST);	calls numargs,DST

Notes: *N* and *X* denote register numbers. *DST* and *SRC* denote any of the addressing modes. *nz* denotes the condition codes; *pc* is the program counter. *REL* denotes any of the six relational operations.

7.3 MINVAX Instruction Set

Addressing Modes

Mode	RTL Notation	Assembly Language
register	r[N]	rN
immediate	C	$C
register deferred	m[r[N]]	(rN)
displacement	m[r[N] + C]	C(rN)

Operations

Operation	RTL Notation	Assembly Language
compare	nz = r[N] ? r[X];	cmp rX,rN
bit test	nz = r[N] & r[X] ? 0;	bit rN,rX
load	r[N] = SRC; nz = SRC ? 0;	mov SRC,rN
store	DST = r[N]; nz = r[N] ? 0;	mov rN,DST
addition	r[N] = r[N] + r[X]; nz = r[N] + r[X] ? 0;	add rX,rN
subtraction	r[N] = r[N] - r[X]; nz = r[N] - r[X] ? 0;	sub rX,rN
multiplication	r[N] = r[N] * r[X]; nz = r[N] * r[X] ? 0;	mul rX,rN
division	r[N] = r[N] / r[X]; nz = r[N] / r[X] ? 0;	div rX,rN
negate	r[N] = -r[N]; nz = -r[N] ? 0;	mneg rN,rN
complement	r[N] = ~r[N]; nz = ~r[N] ? 0;	mcom rN,rN
convert	r[N] = cvt(r[X]); nz = cvt(r[X]) ? 0;	cvt rX,rN
bit set	r[N] = r[N] | r[X]; nz = r[N] | r[X] ? 0;	bis rX,rN
bit clear	r[N] = r[N] & ~r[X]; nz = r[N] & ~r[X] ? 0;	bic rX,rN
shift	r[N] = r[N] << r[X]; nz = r[N] << r[X] ? 0;	ashl rX,rN,rN
exclusive or	r[N] = r[N] ^ r[X]; nz = r[N] ^ r[X] ? 0;	xor rX,rN
jump	pc = LABEL;	jbr LABEL
conditional jumps	pc = nz REL 0 -> LABEL | pc;	jREL LABEL
call instruction (stack)	PC = calls(numargs,DST);	calls numargs,DST

Notes: *N* and *X* denote register numbers. *DST* and *SRC* denote any of the addressing modes. *nz* denotes the condition codes; *pc* is the program counter. *REL* denotes any of the six relational operations.