Ease:
An Environment for Architecture Study and Experimentation

Jack W. Davidson

David B. Whalley

Computer Science Technical Report TR89-08
September 15, 1989

Ease: An Environment for Architecture Study and Experimentation

ABSTRACT

Obtaining accurate and meaningful measurements of the execution behavior of instruction set architectures is difficult. Measurement systems often impose a heavy performance penalty that limits the types of programs that can be used. Furthermore, meaningful measurements can only be obtained if realistic (not toy) benchmark programs are used. Such programs, however, requires high-level language compilers to translate the benchmarks to machine code. The problem is further complicated as most machines require "optimizing" compilers in order for their capabilities to be fully exploited. This paper describes an environment that provides these capabilities. It allows detailed measurements of the execution behavior of realistic programs to be generated without incurring a heavy performance penalty. The accompanying compiler technology allows optimizing compilers to be built that fully exploit the capabilities of the machine. To illustrate the power of the environment, its use in a number of applications is described.

INTRODUCTION

Gathering detailed measurements of the execution behavior of an instruction set architecture is difficult. There are two major problems that must be solved. First, for meaningful measurements to be obtained, programs that represent typical workloads and instruction mixes must be used. This means that high-level language compilers for the target architecture are required. This problem is further compounded as most architectures require an optimizing compiler to fully exploit their capabilities. Building such a compiler is a formidable task in itself.

A second problem is that gathering detailed dynamic measurements of an architecture using typical user programs reading typical data sets can be quite time-consuming. For example, a popular way to gather execution measurements is to simulate the architecture. This is often used when the architecture in question does not yet exist, or is not yet stable and available for production use. Depending on the level of the simulation, programs can run 100 to 500 times slower than directly-executed code [HLT87]. If the architecture being measured exists and is accessible, another popular technique is to instrument the generated code by inserting code to count the number of times instructions are executed. Done naively this can result in programs that run 10 to 20 times slower than the code that is not instrumented. Because of this large performance penalty, the tendency is to use small programs with small data sets. The relevance of measures collected this way is always subject to question.

This paper describes an environment called *ease* (Environment for Architecture Study and Experimentation) that solves both these problems. It consists of a easily retargetable optimizing compiler that produces production-quality code. The compiler also supports the generation of instrumented code that gathers very fine-grained execution statistics. The instrumented code only runs 10 to 15 percent slower than code generated without instrumentation. The combination of an easily retargetable compiler and an efficient method of observing the run-time behavior of real programs provides an environment that is useful in a number of contexts. The paper describes the use of *ease* to evaluate both existing and proposed instruction set architectures, for determining the effect of various machine features, and for determining the best strategies for utilizing machine resources.

ENVIRONMENT FOR OBTAINING MEASUREMENTS

ease logically consists of two parts; the set of tools that permit retargetable optimizing compilers to be built quickly and the tools that produce and analyze the measurements of the execution behavior of the instruction set architecture. The compiler technology is known as *vpo* [BeD88, DaF84, Dav86]. It has been used to build commercial C, Pascal, and Ada compilers. It is the way that the compilers are constructed that lets the measurement system gather detailed measurements of how a particular instruction set architecture behaves at run time. We briefly describe the compiler technology and how its design permits very detailed measurements of the execution behavior of programs to be gathered without substantially increasing the amount of time needed to run the programs being measured.

vpo

The optimizer, *vpo*, replaces the traditional code generator used in many compilers. *vpo* is retargeted by supplying a description of the target machine. Using the diagrammatic notation of Wulf [WJW75], Figure 1 shows the overall structure of a set of compilers constructed using *vpo*. Vertical columns within a box represent logical phases which operate serially. Columns divided horizontally into rows indicate that the subphases of the column may be executed in an arbitrary order. IL is the Intermediate Language. Register transfers or register transfer lists (RTLs) describe the effect of machine instructions and have the form of conventional expressions and assignments over the hardware's storage

cells. For example, the RTL

```
r[1] = r[1] + r[2]; cc = r[1] + r[2] ? 0;
```

represents a register-to-register integer add on many machines. While any particular RTL is machine-specific, the *form* of the RTL is machine-independent. All phases of the compiler manipulate RTLs.

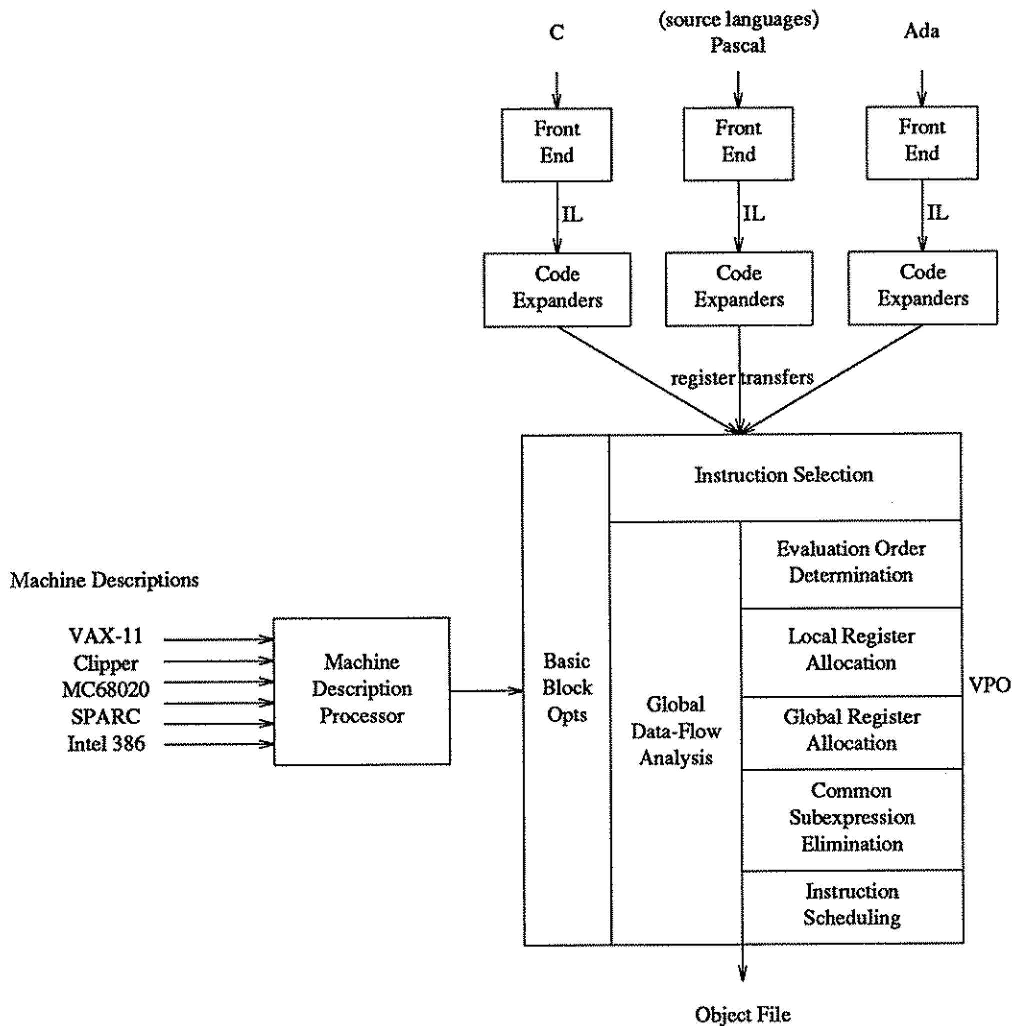

Figure 1: Compiler Structure

There are a number of advantages to using RTLs as the basis of the intermediate representation. Because the form is machine-independent, programs can be constructed that manipulate RTLs in machine-independent ways. For example, the phase that performs data-flow analysis on RTLs is largely machine-independent. Because RTLs represent actual machine instructions, specifics of the target machine are exposed to the various optimization phases resulting in more complete and thorough optimization. Finally, because RTLs are well-defined, it is possible to construct recognizers that can determine whether an RTL represents a legal target machine instruction. The ability to determine, at any time, whether an RTL represents a legal target machine instruction is key to our optimization strategy.

To retarget *vpo* to handle a new architecture, a description of the architecture's instruction set must be written. A machine description consists of a grammar and semantic actions. The grammar is used to produce a parser that checks the syntax of an RTL. The semantic actions check context-sensitive constraints imposed by a particular architecture. Currently, the RTL parsers are constructed using the Unix parser generator *yacc* [Joh78]. There is a certain appeal to the symmetry of using the tool that was used to construct the front end to help construct the back end. Machine description grammars are relatively easy to write [Dav85]. The goal is to compose a grammar and semantic actions that produce a parser that accepts all legal RTLs (instructions) and rejects all illegal RTLs. Our experience is that it is easier to write a machine description for an instruction set than it is to write a grammar for a programming language. The task is further simplified by the similarity of RTLs across machines. This permits a grammar for one machine to be used as the model for a description of another machine. We have used this technique to describe the instruction sets of the following machines: VAX-11, Motorola 68020, National Semiconductor 32016, Concurrent Computer Corporation 3230, Western Electric 32100, Intel 80386, Harris HCX-9, IBM PC/RT, Intergraph Clipper, SUN SPARC, and the AT&T DSP32. The C compiler has been ported to new architectures in as little time as two weeks.

statgen

To be able to evaluate an architecture effectively, one must examine its behavior when executing real programs. To be able to extract this behavior, one must collect measurements from the program's execution that can be used to evaluate the influence of specific features of the architecture. An efficient

way to collect this data for subsequent analysis is to modify the back end of the compiler to store the characteristics of the instructions to be executed and to produce code that will count the number of times that each instruction is executed. These modifications, called *statgen*, have been implemented in the *vpo* compiler system and are described in subsequent sections. The method is illustrated in Figure 2.

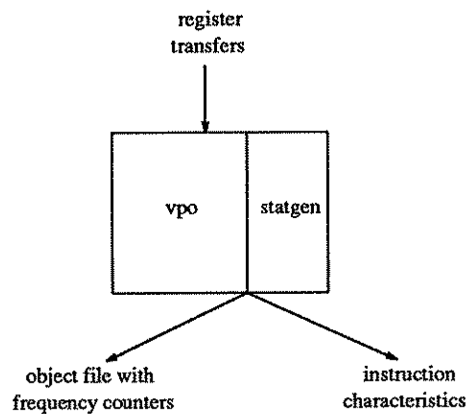

Figure 2: Method for Gathering Measurements

Instruction Characteristics

The first modification to *vpo* necessary to collect measurements is to have the compiler save the characteristics of the instructions that will be executed. As an instruction is parsed, information about the characteristics of the instruction is collected and used for semantic checks. The semantic checks are now modified to store these characteristics with the RTL representing the instruction by invoking a machine-independent routine. The routine is only invoked if the option for collecting data is set and no semantic errors have occurred. The routine receives the instruction type and the semantic record containing the fields of the instruction. After all optimizations have been completed, most of the instructions have been parsed. Those instructions that have not yet been parsed are then parsed and their characteristics stored with the instruction. The information about each instruction is then written to a file. An example of a routine that stores information about a Motorola 68020 call instruction is shown in Figure 3.

```
/*
 * call - check semantics of call
 */
void call(i1)
struct sem_rec *i1;
{
   if (dassem)
      printf("\\tjbsr\\t%s\\n", i1->sem.call.addr->asmb);
   else if (!erflag && swm)
      stinstinfo(JSBRI, i1);
}
```

Figure 3: Storing Instruction Information

Frequency Counters

The second modification is to have the compiler generate code to count the number of times each instruction is executed. This is accomplished after all optimizations have been performed. Within each function there are groups of instructions, basic blocks, that are always executed the same number of times. There are also groups or classes of basic blocks that are executed the same number of times and these are denoted as execution classes. Thus, the code that the compiler generates to count the number of times that each instruction in an execution class is executed is inserted at the beginning of the first basic block in the execution class.

An example of inserting frequency counters is given in Figures 4 through 7. Figure 4 contains a C function. Figure 5 gives the VAX-11 assembly code that would normally be produced by *vpo* for that C function. Figure 6 shows the same assembly code broken into basic blocks. Note that although there are five basic blocks there are only three execution classes ($\{1,5\}$, $\{2,4\}$, $\{3\}$). Figure 7 shows the modified Vax assembly code with execution class counters inserted. The name of the file being compiled, *test* in this case, is used to distinguish counters from other files in the same executable.

Determining whether a block belongs to an execution class is done in three steps. First, the set of blocks that dominate the current block must be calculated. This information is already available in *vpo* if the option to allocate variables to registers has been set. The second step determines if the current block is always a successor to the blocks within an execution class. This is accomplished by determining if all paths from one block eventually lead to the current block. The third step checks if the current block is in

the same set of loops as the blocks in the execution class. The information for this step is also already available in *vpo*.

```
int foo(k)
int k;
{
   int i, j[10];

   if (k > 5) {
      for (i = 0; i < 10; i++)
         j[i] = 0;
      k = 2;
      }
   return (k);
}
```

Figure 4: C function

```
.text
.globl _foo
_foo:
.word   0x0
.set    k.,4
.set    j.,-40
        subl2   $40,r14
        cmpl    k.(r12),$5
        jleq    L14
        clrl    r2
L17:    clrl    j.(r13)[r2]
        aoblss  $10,r2,L17
        movl    $2,k.(r12)
L14:    movl    k.(r12),r0
        ret
.data
```

Figure 5: Vax Assembly Code for Function in Figure 4

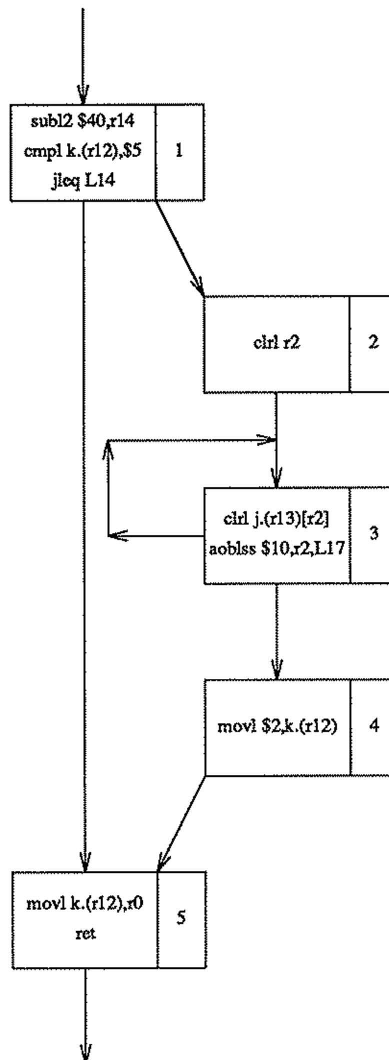

Figure 6: Assembly Code of Figure 5 in Basic Blocks

There is little overhead for collecting data to produce the measurements. For instance, on the VAX-11 the C benchmarks `whetstone` and `dhrystone` were executed with and without data collection. The two benchmarks executed with data collection code inserted required only 6% and 13% respectively more execution time than they required without data collection instructions. The number of counters needed for `whetstone` was fifty-one. If execution classes were not used and counters were placed at each basic block, `whetstone` would have required eighty-four counters.

```
.text
.globl _foo
_foo:
.word   0x0
.set    k.,4
.set    j.,-40
        incl    _test_counts
        subl2   $40,r14
        cmpl    k.(r12),$5
        jleq    L14
        incl    (_test_counts + 4)
        clrl    r2
L17:    incl    (_test_counts + 8)
        clrl    j.(r13)[r2]
        aoblss  $10,r2,L17
        movl    $2,k.(r12)
L14:    movl    k.(r12),r0
        ret
.data
```

Figure 7: Vax Assembly Code with Frequency Counters

Other Modifications

Some types of measurements require additional code to be generated. For instance, the number of times that conditional branches are taken can be measured by inserting code after each conditional branch to count the number of times each conditional branch was not taken. This count can then be subtracted from the number of times that conditional branches are executed to produce the desired measurement. Other measurements, such as a trace of addresses of instructions or frequencies of pairs of instructions, require knowledge of the sequence of blocks that are executed. For these types of measurements a routine written in a high-level language can be invoked at the beginning of each basic block with the current block number accessible.

Processing the Collected Data

The data that is collected can be stored and then analyzed at a later time. Separating the collection and analysis of measurements has a number of advantages. If different evaluations of the data are

required, then collection of the data is required only once. If analysis of the execution of several different programs is needed, then the data can be collected from each program's execution separately. Finally, the analysis of the data is separated from the generation of the data and thus requires fewer modifications to the back end of the compiler.

At the end of the execution of the program, the number of times that each execution class is executed is written to a file. The execution counts and the characteristics of the instructions will then both be used to produce the dynamic measurements. The characteristics of the instructions can also be used to produce the static measurements. Figure 8 shows how both static and dynamic measurements can be obtained.

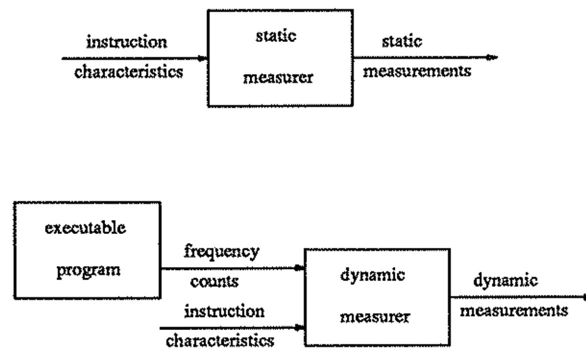

Figure 8: Producing Reports

APPLICATIONS

ease can be used for several different applications which include:

1. evaluation of existing architectures
2. design of new architectures
3. analysis of code generation strategies
4. automatic optimization of programs

Evaluation of Existing Architectures

Past architectural studies have suffered from many limitations. Some used a small set of benchmark programs due to the difficulty of collecting data. For instance, in the CFA architecture evaluations [FSB77], twelve assembly language programs were used to evaluate and rank nine different architectures. Most of these programs were less than 200 static machine instructions. Wiecek gathered dynamic measurements of VAX-11 instruction set usage from the execution of six different compilers [Wie82]. Because she used trace software to trap on each instruction to generate the trace data, each program compiled was small to allow measurements to be collected in a reasonable amount of time.

Many studies that compare architectures do not account for differences in how the machine instructions are produced. Each test program in the CFA architectural evaluations was hand-coded in the assembly language of the machine to test a specific feature of an architecture [FuB77]. Thus, the quality of the test programs depended upon the skill of the programmer and his knowledge of the machine. Patterson claimed that the different compilers used in his study used the same compiler technology [PaP82]. The Portable C Compiler (*pcc*) [Joh79] was ported to the VAX-11 and RISC I. However, the quality of the code produced by each *pcc* compiler depends on the skill of the compiler writer when constructing tables for code generation and the patterns for peephole optimization.

This research has eliminated these problems. Using the previously described method for collecting data, one can use a number of realistic programs and collect the data in a timely fashion. For instance, on the VAX-11/8600 measurements can be collected from the execution of almost 100 million instructions in less than ten minutes. Since the code selector and other optimizations are constructed automatically, the quality of the code generated by *vpo* for each of the architectures has less dependence on the skill of the implementors than compilers using other techniques. Retargeting the compiler to a new machine only requires describing the architecture. Ad hoc case analysis is unnecessary. Thus, the programs compiled for each of the architectures receive the same degree of optimization.

ease has been ported to ten different machines to compare current architectures. These machines listed below include three CISCs and three RISCs.

1. VAX-11 — DEC VAX-11/8600
2. HCX — Harris HCX-9
3. 3B15 — Western Electric 32100
4. 68020 — Motorola 68020/68881
5. 32016 — National Semiconductor 32016
6. 80386 — Intel 80386/80387
7. 3230 — Concurrent 3230
8. RT — IBM PC/RT
9. CLIP — Intergraph Clipper
10. SPARC — SUN SPARC

Measurements from the execution of a test set of nineteen C programs were obtained for each of the architectures. The detail and accuracy of the reports produced by *ease* allowed more insights to be drawn when analyzing the measurements. For instance, Figure 9 shows the number of times memory was referenced for data (not instruction fetches) by each machine. The number of memory references due to referencing variables and spills of temporaries is shown in solid lines. The additional number of memory references due to saving and restoring allocable registers is shown in dashed lines. The additional number of memory references due to handling function linkage (stack pointer, frame pointer, program counter, etc.) is shown in dotted lines. Thus, *ease* not only determines the total number of memory references, but also the reason for each memory reference. It is interesting to note that about 25% of the VAX-11 memory references is due to function linkage. This results in functions calls on the VAX-11 being very expensive. The SPARC architecture with its registers windows, however, had very few memory references due to saving and restoring registers or function linkage.

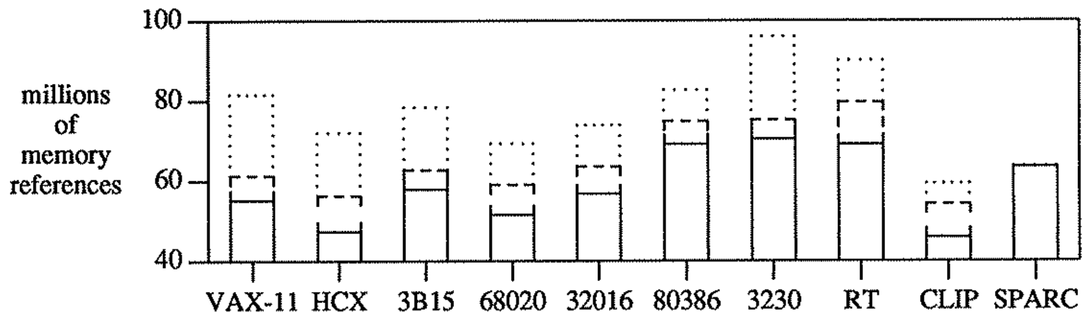

Figure 9: Number of Memory References

There are many other detailed measurements that were obtained. The measurements collected include:

1. instruction path length
2. instruction path size
3. instruction type distribution
4. addressing mode distribution
5. memory reference size distribution
6. memory reference address distribution
7. register usage
8. condition code usage
9. conditional branches taken
10. average number of instruction between branches
11. data type distribution

The measurements were sufficiently detailed to determine the number of times each combination of addressing mode and data type were used for each field of each type of instruction. Results comparing these architectures appears in Wha89.

Design of New Architectures or Architectural Features

In addition to using *ease* to evaluate and analyze existing instruction set architectures, it can be used to help design new machines. There are two problems, however. First, because the machine does not exist, there is no good way to execute code and see how it performs or to compare different instruction sets. The solution often used is to construct a simulator for the proposed instruction set. A simulator imitates a machine by interpreting the machine instructions on a host machine [AlW75,BSG77]. The problem is that these simulators run too slow to be used with real programs. Depending on the level of simulation, execution times hundreds of times slower than direct execution can be expected.

A second problem is that compilation issues play an important role in the design of a new machine. Better computer systems (hardware and software) are possible if the architecture is designed to operate synergistically with the compiler. As examples we point to the IBM 801 [Rad82] and the MIPS processor [HJB82]. Their designs were influenced to a large degree by the decision to make pervasive use of high-level languages and powerful compilers.

ease soves both these problems. The compiler problem is largely solved by *vpo* and the accompanying front ends. A full compiler for a new architecture can be constructed in a few weeks. The

problem of slow execution of programs is solved by having *vpo* emit code for an existing host machine that emulates the instruction set of the machine being designed. *vpo*'s organization permits this to be done quickly and easily as follows.

The last step in the compilation process is the conversion of an RTL to assembly language for the target machine and emitting it to a file that will be processed by the system's assembler. In order to evaluate an architecture that does not exist, instead of emitting assembly language for the target machine, assembly code for an existing architecture is emitted. Information about the effects of the instruction are emitted as if the target architecture existed. Figure 10 contains the code that allows the VAX-11 increment instruction to be evaluated on the SPARC.

ease can also be used to measure the influence of adding new architectural features to an existing machine. For instance, the number of available registers was increased on the VAX-11 architecture to

```
/*
 * binst - check semantics of binary operation
 */
void binst(i1)
struct sem_rec *i1;
{
    .
    .
    /* Emit an inc inst if an add and the increment is 1 */
    if (t->op == '+' && strcmp(t->sem.binsti.right->asmb, "$1") == 0)
       if (vaxassem)
          printf("\tinc%c\t%s\n", typech2(t), i1->sem.binsti.dst->asmb);
       else if (sparcassem)
          if (MEM(t->sem.binsti.dst)) {  /* g7 extra register */
             printf("\tld\t%s,%%g7\n", i1->sem.binsti.dst->asmb);
             printf("\tadd\t%%g7,1,%%g7\n");
             printf("\tst\t%%g7,%s\n", i1->sem.binsti.dst->asmb);
             }
          else
             printf("\tadd\t%s,1,%s\n", i1->sem.binsti.dst->asmb);
    .
    .
}
```

Figure 10: Code to Generate an Increment Instruction

determine its effect. First a set of currently available registers, equal to in number to the maximum number of unique registers that could be used in one instruction, were reserved. Any references to the reserved or new registers were replaced by corresponding memory references. If one of these registers was referenced in an addressing mode, then the value for the register was loaded from memory into a reserved register previous to the instruction. The reserved register, instead of the memory reference, was then used in the instruction. If one of these registers was updated as a side-effect of using the addressing mode, then the new value for the register was stored after the instruction. These updates were accomplished by adding less than fifty lines of code. An example of translating VAX-11 RTLs referencing additional registers is shown in Figure 11.

```
r[31] = r[31] + 1;      /* RTL referencing a non-existent register */

=>

incl    R31(fp)         /* becomes memory reference */


R[a[r[31]++]] = 0;      /* RTL referencing a non-existent register */

=>

movl    R31(fp),r6      /* becomes a sequence of instructions */
clrl    (r6)+
movl    r6,R31(fp)
```

Figure 11: Translating Additional Registers

Analysis of Code Generation Strategies

ease has also been used to evaluate calling sequence conventions. By recompiling the source files from the C run-time library, different calling sequence conventions can be tried. By extracting measurements of the behavior of the code, the effect of any change can be easily observed. For example, one experiment we tried was varying the partitioning of scratch and non-scratch registers in a callee-save calling sequence. Using execution times the effects of using different combinations of scratch and non-

scratch registers was not obvious. However, *ease* was able to accurately reveal the results of the changes as shown in Figures 12 through 14. These figures show the effects of changing the partitioning of the scratch and non-scratch sets of registers on the total instructions, memory references, and static code size on the VAX-11. These results imply that an inappropriate partitioning may result in poorer performance.

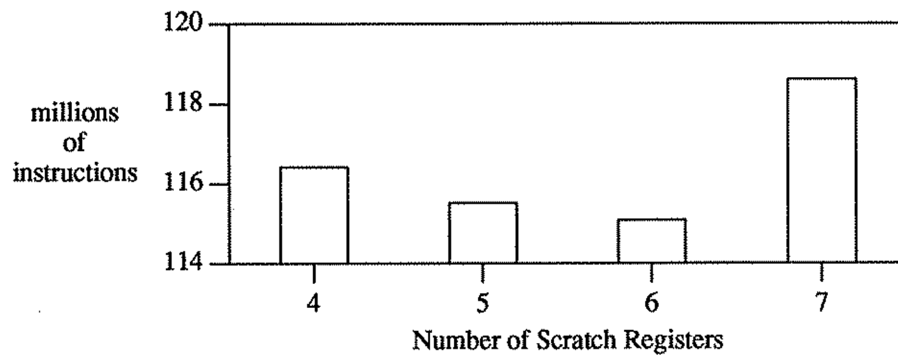

Figure 12: VAX-11 Total Instructions

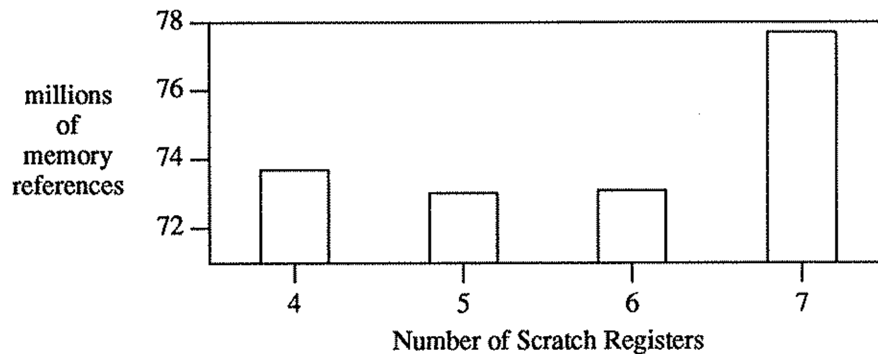

Figure 13: VAX-11 Total Memory References

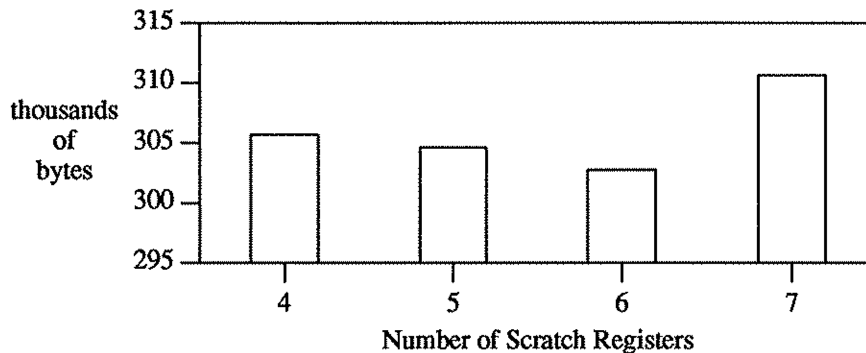

Figure 14: VAX-11 Code Size

ease has also been used to perform experiments with other calling sequence conventions. For instance, five different methods for saving and restoring registers without interprocedural analysis or special hardware have been examined. The mechanism used to pass arguments to a function is another calling sequence convention. The benefits of passing arguments through registers as opposed to passing all arguments on the stack have been analyzed. Also the use of primitive call and return instructions has been compared with the use of their more complex counterpart instructions.

Other Applications

ease can be used in other ways. Because *ease* gathers detailed information about a program's behavior with little overhead, it is possible that this information can be used to automatically optimize a program. Most optimizers estimate usage of resources in order to determine what transformations to make to a program. For example, estimates of the number of references to variables are used by the optimizer to decide which variables should be placed in registers. Those variables with the highest estimate reference cost are allocated first. Other decisions by the optimizer, such as whether to move code out of loops, expand function calls inline [DaH88], and how to structure loops are also based on estimates of the program's behavior.

While these estimates work reasonably well, we envision a system where profile data for a program collected by *ease* can be fed back into *vpo*. Rather than using estimates, the optimizer will have detailed information about the program's behavior. Assuming that the programmer uses realistic data to

collect the execution statistics, such a system should substantially improve the run-time performance of programs. In some sense, the programs are being "hand-tuned" automatically. We are currently constructing a prototype of this system to determine whether it is effective.

We have also used *ease* to contrast static and dynamic measurements. By using the measurements as input to a statistical package, strong linear relationships were found between most of the static and dynamic measurements [DRW89]. Typically static measurements are easier to obtain and dynamic measurements give more useful information on performance. Regression analysis was used to produce an equation that estimates each dynamic measurement from its corresponding static measurement.

CONCLUSIONS

This environment for the collection of architectural measurements has been designed to require little effort when retargeting for a new architecture. Since the code selector and other optimizations are constructed automatically, the *vpo* compiler system is easy to retarget. The semantic record constructed from parsing an RTL representing an instruction is used to produce assembly code for proposed and existing architectures and to store instruction information for the collection of measurements. Most of the code to perform the extraction of measurements has also been accomplished in a machine-independent fashion. The *vpo* compiler for ten different machines was modified to collect measurements as specified above. It typically took three or four hours to make the machine-dependent modifications for the compiler on each machine.

There is not much overhead for collecting data to produce measurements. Typically, the instrumented code ran 10 to 15 percent slower than code that was not instrumented. Because the information about instructions is collected as a side effect of the compiler parsing instructions, *ease* also only required 15 to 20 percent overhead in compilation time.

The *ease* environment has been shown to be an efficient tool for architectural evaluation and design. Since accurate and detailed reports can be produced for a variety of measurements, the impact of each modification to the compiler or architecture can easily be determined. This allows one to use an iterative design method for evaluation of performance in a quantitative manner.

ACKNOWLEDGEMENTS

Manuel Benitez implemented the machine-independent portion of the *vpo* compiler system.

REFERENCES

[AlW75] W. G. Alexander and D. B. Wortman, Static and Dynamic Characteristics of XPL Programs, *Computer 8*,11 (November 1975), 41-46.

[BSG77] M. R. Barbacci, D. Siewiorek, R. Gordon, R. Howbrigg and S. Zuckerman, An Architectural Research Facility—ISP Descriptions, Simulation, Data Collection, *Proceedings of the AFIPS Conference*, Dallas, TX, June 1977, 161-173.

[BeD88] M. E. Benitez and J. W. Davidson, A Portable Global Optimizer and Linker, *Proceedings of the SIGPLAN Notices '88 Symposium on Programming Language Design and Implementation*, Atlanta, GA, June 1988, 329-338.

[DaF84] J. W. Davidson and C. W. Fraser, Code Selection through Object Code Optimization, *Transactions on Programming Languages and Systems 6*,4 (October 1984), 7-32.

[Dav85] J. W. Davidson, Simple Machine Description Grammars, TR85-22, University of Virginia, November 1985.

[Dav86] J. W. Davidson, A Retargetable Instruction Reorganizer, *Proceedings of the SIGPLAN Notices '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986, 234-241.

[DaH88] J. Davidson and A. Holler, A Study of a C Function Inliner, *Software—Practice & Experience 18*,8 (August 1988), 775-790.

[DRW89] J. Davidson, J. Rabung and D. Whalley, Relating Static and Dynamic Measurements, Technical Report 89-03, University of Virginia, Charlottesville, VA, July 1989.

[FSB77] S. H. Fuller, H. S. Stone and W. E. Burr, Initial Selection and Screening of the CFA Candidate Computer Architectures, *Proceedings of the AFIPS Conference*, Dallas, TX, June 1977, 139-146.

[FuB77] S. H. Fuller and W. E. Burr, Measurement and Evaluation of Alternative Computer Architectures, *IEEE Computer 10*,10 (October 1977), 24-35.

[HJB82] J. Hennessy, N. Jouppi, F. Baskett, T. Gross and J. Gill, Hardware/Software Tradeoffs for Increased Performance, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, March 1982, 2-11.

[HLT87] M. Huguet, T. Lang and Y. Tamir, A Block-and-Actions Generator as an Alternative to a Simulator for Collecting Architecture Measurements, *Proceedings of the ACM SIGPLAN Notices '87 Symposium on Interpreters and Interpretive Techniques*, St. Paul, Minnesota, June 1987, 14-25.

[Joh78] S. C. Johnson, Yacc: Yet Another Compiler-Compiler, *Unix Programmer's Manual 2B*,19 (July 1978), 1-34.

[Joh79] S. C. Johnson, A Tour Through the Portable C Compiler, *Unix Programmer's Manual, 7th Edition 2B*(January 1979), Section 33.

[PaP82] D. A. Patterson and R. S. Piepho, RISC Assessment: A High-level Language Experiment, *Proceedings of the Ninth Annual Symposium on Computer Architecture*, Austin, TX, April 1982, 3-8.

[Rad82] G. Radin, The 801 Minicomputer, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, March 1982, 39-47.

[Wha89] D. Whalley, *A Study of High-Level Language Architectures*, PhD Dissertation Proposal, University of Virginia, Charlottesville, VA, 1989.

[Wie82] C. A. Wiecek, A Case Study of VAX-11 Instruction Set Usage for Compiler Execution, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, March, 1982, 177-184.

[WJW75] W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs and C. M. Geschke, *The Design of an Optimizing Compiler*, American Elsevier, New York, NY, 1975.