CHAPTER 1

INTRODUCTION

This dissertation describes an environment for the evaluation of computer architectures and architectural features. The first chapter provides background information for the problem of obtaining program measurements. Initially, limitations of past architectural studies are described and the goals for the dissertation are stated. This is followed by a review of the area of obtaining program measurements. First, the different classes of program measurements are characterized. Next, the methods used in the past for obtaining program measurements are described and their strengths and weaknesses are identified. The applications of analyzing program measurements are then examined and past studies using program measurements and their contributions are discussed.

The second chapter describes the environment that was constructed for obtaining architectural-level measurements. The modifications to an existing retargetable optimizer are illustrated and the manner in which reports are generated from the information collected is shown.

The third chapter describes an architectural study performed with this environment that analyzed measurements collected from the execution of the same set of test programs on each machine. First, the different kinds of measurements extracted from each architecture are discussed, background information about each of the architectures in the study is given, and the set of test programs used in the study is specified. The measurements obtained from the architectures are then analyzed. The dynamic measurements are first examined and the implications of this information are discussed. Static measurements are then compared to the corresponding dynamic measurements. Strong linear relationships between most of the static and dynamic measurements were discovered. Statistical methods were used to produce equations to estimate each dynamic measurement, which give more useful information on performance, from its corresponding static measurement, which are easier to obtain. The last section in this chapter compares each of the architectures by attempting to determine the total cost of the execution of the test set.

The fourth chapter describes some interesting experiments performed with the environment. The first experiment analyzed six different methods for saving and restoring registers. In one set three of the schemes do not use data flow analysis to minimize the number of saves and restores, while the three schemes in the other set do. Within each set a callee save, caller save, and hybrid approach are used. The next experiment evaluates the effectiveness of passing arguments in registers as a calling sequence convention. The following experiment abandons complex call and return instructions in favor of more primitive instructions. The combination of passing arguments through registers and using more primitive instructions allowed new optimizations to be accomplished on a significant percentage of the executed calls. The last experiment analyzed the effectiveness of a new technique for reducing the cost of branches by using registers. This technique has the advantages of reducing the number of instructions executed, eliminating many pipeline delays, and potentially decreasing the delay due to instruction cache misses.

The final chapter gives conclusions for the dissertation. The achievement of the goals of the dissertation and contributions of this research are discussed.

1.1. Motivation

To effectively evaluate an existing or proposed computer architecture, one must analyze measurements from typical programs that are to be executed by the machine. The analysis of these measurements can be used to determine the influence of a specific feature on the effectiveness of the architecture and to compare different architectures. The method used to extract these measurements can impact the accuracy and quality of architectural evaluations.

Past architectural studies have suffered from one or more limitations due to the methods used for collecting program measurements. Methods such as simulation and tracing are very time-consuming and the use of these methods can result in studies analyzing only a small number of executed instructions. Some studies have compared different architectures without consideration of the compiler for each machine. Since the quality of the code produced by different compilers can vary, these studies are comparing compilers along with architectures. The implementation of several methods for the extraction of program measurements can require much effort and time. For instance, the effort to construct a simulator is comparable to the effort to construct a compiler. The difficulty of implementing these methods may discourage one from collecting program measurements for architectural evaluations. The extraction of specific measurements using some methods can be difficult or impossible. Lack of detailed information can lead to assumptions that may be inaccurate. Many methods prohibit experimentation with an architectural feature or an architecture that does not yet exist. Such experimentation is vital since it produces empirical results that can guide architectural design. Thus, some limitations of previous architectural studies include:

1. a small set of benchmark programs
2. differences in how machine instructions are produced
3. excessive effort required to implement a method to extract measurements
4. inability to obtain certain types of measurements
5. difficulty in performing experiments

The last major architectural study that evaluated several different architectures, the CFA evaluations [FuB77], was performed over thirteen years ago. The CFA evaluations collected a few detailed measurements from a set of small programs. Many current architectures are now quite different from the architectures in the previous decade. A thorough evaluation of several current architectures would provide valuable information to computer architects for modification of existing machines and design of new ones.

An environment that quickly collects accurate and detailed dynamic measurements from programs and facilitates experimentation can aid in the interactions between the design of an architecture and a compiler. An architecture is often designed first without determining if a compiler can make effective use of the features of the architecture. One area that typifies this problem is the calling sequence conventions imposed by the writer of the compiler and run-time library. These conventions often seem to be imposed as an afterthought with little analysis and are typically restricted by the available instructions on the machine for implementing function calls and returns. Using an environment that facilitates the joint development of an architecture and compiler can result in a more effective implementation.

There are three goals for this research. The first goal is to provide an environment in which static and dynamic architectural-level program measurements can be collected without the problems of past architectural studies. These program measurements include the frequency of occurrence of instructions and addressing modes, program size information, and register usage. A more detailed description of the types of measurements is given in a later section. The second goal is to use this environment to perform an architectural study on several current architectures. This study analyzes the dynamic measurements collected from each of the architectures and examine the differences between static and dynamic measurements. The last goal is to use the environment to perform experiments in code generation and machine design. These experiments not only provide insight on the influence of some architectural features and conventions, but also demonstrate the effectiveness and utility of the environment and suggest areas in which the environment can be extended.

1.2. Review

Classes of program measurements, different methods of obtaining program measurements, applications of analyzing program measurements, and previous studies using program measurements are examined in the following sections.

1.2.1. Classes of Program Measurements

Program measurements are an extraction of some of the quantitative attributes from the programs that are executed on a computer. Classes of program measurements vary depending on two factors. One is the form or level of the program from which the measurements are extracted, while the other is the state of the program when the measurements are obtained.

Programs measurements can be obtained at the source and architectural levels. Measurements obtained from the source code supply information about the usage of the features of the programming language, whereas, measurements obtained from the assembly or object code provide information about the usage of the features of the architecture. Source-level measurements can be correlated to architectural-level measurements. Thus, source-level measurements can be used to provide insight for architectural decisions on any machine with a compiler for that programming language. This correlation, however, is not always obvious. Architectural-level measurements give more precise information about a specific architecture.

Two states of programs in which measurements can be obtained are the static and dynamic states. Static measurements refer to the text of a program and count each statement or machine instruction equally. Dynamic measurements refer to the execution of a program and count each statement or machine instruction by the number of times that it is executed. Static measurements are easier to obtain and guarantee complete coverage of the program. Since each statement or machine instruction in a program is rarely executed the same number of times, static measurements do not give realistic information on performance. The dynamic coverage of the program, however, is driven by the test data. If the test data is not representative of how the program will be used on the machine, then the dynamic measurements are less meaningful.

1.2.2. Methods of Obtaining Program Measurements

There have been several different methods or combination of methods used to collect program measurements. The factors that may influence the choice of a method include:

1. implementation effort
2. accuracy of measurements
3. types of measurements required
4. computational requirements for obtaining measurements

Several techniques for collecting program measurements are described below.

Program simulation imitates a machine by interpreting the machine instructions [AlW75, BSG77]. Dynamic architectural-level measurements are updated as the simulator interprets each instruction. The efforts to construct a simulator and a compiler are comparable. The total execution time of a simulated program is typically hundreds of times slower than if the program is executed directly [HLT87]. The main advantage of program simulation is its portability. Since a simulator for a specific architecture can be written in a high-level language, it can be executed on any machine that supports that language. Thus, measurements can be obtained for a machine that does not yet exist.

Program tracing records a sequence of events which occur during the execution of a program. These events may include the sequence of instructions executed and the memory addresses referenced. Program tracing can be accomplished by several mechanisms and is sometimes available as an option on a machine. One way to implement tracing is to have the machine interrupted after the execution of each instruction [DDD81]. The interrupt handler then invokes a user-defined handler that records the trace information associated with the instruction, which can produce large volumes of output. A filter, a program that reads input data, performs some operation on that input, and writes output data, can process the trace output to obtain the program measurements. The execution of a program with tracing enabled may run 1000 times slower than one without tracing [HLT87]. An advantage of using tracing is the ability to collect dynamic architectural-level measurements that require knowledge of the order of the execution of instructions [Lun77, PeS77, Wie82].

Program sampling periodically interrupts a program for a very short interval of time in order to inspect its state. Measurements are collected during each of these intervals. Sampling incurs little overhead since the program being measured is only interrupted for a small percentage of its execution time. Sampling measurements typically vary with different executions of the same program and thus can only be used as estimates of complete dynamic measurements. If the state of the program is sampled more frequently, the accuracy of the measurements improve and the overhead of obtaining measurements increases. Profiling is the dynamic measurement of the frequency or time spent at each place in a specific program. The profiling facility of UNIX uses sampling to estimate the execution time required by each routine in a program [GKM82].

Program monitoring collects dynamic architectural-level program measurements without perturbation of a machine. This method is only possible with a hardware device known as a hardware monitor. The frequency and time taken by each instruction can be measured with this method [ClL82]. The advantage of this method is that program measurements can be collected with no overhead in execution time. The disadvantages include limited measurements and the requirement of a specialized, expensive, hardware device.

Program analysis, a simple and fast method, extracts static measurements directly from the text of a program. Filters can easily be constructed by modifying the parser of a compiler or an assembler to obtain the measurements. Since this method just collects static measurements, more realistic information on performance can only be obtained using other methods that collect dynamic measurements [Kat83]. This method has been used to extract static measurements from both source and machine code [Coo89, Dit80, Knu71, SwS82].

Program instrumentation modifies a program to increment counters during execution. Static information, collected by program analysis, is associated with each counter. After the termination of the execution of the program, each set of static information is weighted by the value of its associated counter to produce the dynamic measurements. This method has been used to modify the source code of a program to obtain dynamic source-level measurements [Knu71]. It has also been used to modify assembly or machine code to obtain both dynamic source-level measurements, if optimizations are not applied across source-level statement boundaries [Tan78], and dynamic architectural-level measurements [HLT87]. Modification of programs has been accomplished through the use of filters or as an option in compilation. Since a basic block has only one entry and one exit point, the instructions within a basic block are always executed the same number of times. Counting the number of times that each basic block is executed to measure the frequency of executed instructions was proposed by Weinberger [Wei84]. Modification captures program measurements with little overhead in execution time. One must ensure, however, that the modifications made to collect measurements do not change the execution behavior of the program.

An emulator has been defined to be a complete set of microprograms which, when embedded in a control store, define a machine [Ros69]. *Emulator instrumentation* modifies the microcode of a machine to update dynamic architectural-level measurements after the execution of each macroinstruction. This method does not require much runtime overhead as compared to tracing or simulation. For example, modification to collect instruction frequency information slowed a Mesa emulator by about a factor of six [McD82]. Another problem is that the microcode of a machine is not always accessible to one desiring to collect program measurements. Even if it can be accessed, microcode is also difficult to modify and maintain which would discourage experimentation.

1.2.3. Applications of Analyzing Program Measurements

Program measurements have been used in architectural applications for the evaluation and modification of existing architectures and the design of new architectures. The effectiveness of instructions and addressing modes has been evaluated by examining the frequency in which they occur [ClL82, Dit80, SwS82, Wie82]. Frequency measurements can be used with timing information to provide a more accurate estimate of the influence of an instruction or addressing mode on the performance of a machine [PeS77]. Frequency of occurrence of instructions and addressing modes has also been used as the criteria for encoding instructions [CoD82, Tan78]. Traces of addresses have been used for cache and paging design [Win73]. Evaluating dynamic sequences of instructions has been used to propose new instructions, guide pipeline design, and evaluate instruction buffer size [Lun77, McD82, PeS77, SwS82]. By analyzing program measurements one can use an iterative design method to evaluate the effectiveness of modifications such as adding a new instruction or changing the number of registers.

Program measurements can be used in several applications in the area of compilers. Frequency information can indicate if the compiler is not able to find specific instructions or addressing modes. Measurements indicating the costs of operations and operands can aid in the evaluation of different compilers. These measurements can also help the compiler writer evaluate the effectiveness of different optimizations. Measurements of the frequency spent at each place in a program have been redirected as input to the compiler as an aid in compiler optimization [CNO87].

1.2.4. Previous Studies Using Program Measurements

There have been many studies involving program measurements. These studies have varied in many aspects including the set of benchmark programs used, types of measurements obtained, and method of data collection.

1.2.4.1. Studies Using Source-Level Program Measurements

Knuth performed one of the earliest and most influential studies of source-level program measurements [Knu71]. He took static source-level measurements of 440 FORTRAN programs and dynamic source-level measurements of twenty-four FORTRAN programs. The static measurements were collected by program analysis and the dynamic measurements were collected by program instrumentation of the source code. Both the static and dynamic measurements were used to analyze FORTRAN language feature usage. By examining the static measurements Knuth discovered that most assignment statements were very simple. He found that 68% of assignment statements had no operators on the right-hand side and 17.5% had only one operator. Of those assignments with one or more operators, 39% had the same variable being assigned the value of the expression as the first operand in the expression. Knuth also discovered that 58% of static variable references were simple scalars. Analysis of the dynamic measurements showed that 67% of all statements executed were assignment statements. Since a large percentage of the assignment statements had no operators on the right-hand side, this result implies that special attention should be given simple data movement instructions on a machine.

Alexander and Wortman [AlW75] collected static and dynamic source-level measurements from ten XPL programs on a IBM/360. Program analysis was used to collect the static measurements by modifying the compiler to count the number of times that each feature of the language was used. Using the static measurements and interpretive execution they were able to produce the dynamic measurements. The measurements collected were used to analyze XPL language feature usage. Analysis of dynamic measurements revealed that 42% of the XPL statements executed were assignment statements, 13% were conditional statements, and 13% were call statements. This showed that a large percentage of instructions dealt with moving data and that transfers of control occurred frequently. They discovered that 72% of compiler productions applied were used to parse expressions as opposed to parsing other constructs in the XPL language. As a result of this finding, several compiler writers use a different technique for parsing expressions to increase the speed of compilation [Flo63, Han85]. Alexander and Wortman also used the measurements to analyze the IBM/360 instruction set. They discovered that 56% of all numeric constants could be represented in four bits and that the branch destination of over half the branching instructions executed was no more than 128 bytes distant. These observations imply that short forms of constants and branch address destinations may be effective for reducing the average size of an instruction.

Tanenbaum [Tan78] collected static and dynamic source-level measurements from more than 300 procedures written in SAL on a PDP-11/45. Program analysis was used to collect static measurements by modifying the compiler to count the number of times each feature of the source language appeared in the program. The collection of dynamic measurements was accomplished using program instrumentation by modifying the compiler to insert instructions in the object code. Using the measurements he analyzed the use of language features in SAL. He discovered that 41.9% of statements executed were assignment statements, 36% were if statements, and 12.4% were call statements. Of the assignment statements executed, 19.2% stored a constant and 66.3% had no operator on the right side of the assignment. Tanenbaum found that the addition operator accounted for 57.4% of arithmetic operators used and the comparison for equality accounted for 50.6% of relational operators used. Analysis of procedure calls revealed that 72.1% of the calls executed had two or less arguments. The static measurements also showed that there were many simple operations that occurred frequently. Given all other factors as being equal, a smaller program will run faster than a larger one since fewer bits are fetched from memory. A smaller program can also reduce the number of page faults and cache misses in a memory hierarchy. The results from the analysis were used to propose a new instruction set and how the instruction set should be encoded. By encoding the frequently occurring instructions, he reduced program size by a factor of three.

Cook and Lee [CoL82] collected static measurements using program analysis from more than 120,000 lines of Pascal programs. A filter was constructed by modifying a Pascal parser to collect these measurements in twelve different contexts. The contexts included procedures, functions, various control statements, conditional expressions, left and right sides of assignment statements, and argument lists. They discovered that less than 12% of subprograms were nested inside other subprograms. Analysis of the measurements also revealed that global variables in enclosing procedures were referenced infrequently. These results showed that implementing an expensive mechanism to perform up-level frame addressing efficiently would not be worthwhile. An investigation of argument passing indicated that 84% of all procedures had fewer than four local variables, 97% had fewer than four arguments, and 67% of arguments passed to routines were simple variables and constants. The results of this investigation implies that short forms of displacement addressing modes would be effectively used and the importance of a simple `push` instruction. Cook and Lee discovered that constants accounted for 34% of all operands. A large percentage of operands as constants signifies the importance of having an immediate addressing mode. They found that 86% of `for` loops contained less than eight instructions. A small number of instructions in `for` loops indicates that the iteration variable of the `for` loop could be kept in a register since the register could be reused in nonoverlapping `for` loops and that an instruction buffer may be effective. An examination of the distribution of integer constants revealed that negative constants appeared much less frequently than positive constants. Such a skewed distribution suggests that implementing immediate addressing to allow a symmetric range of negative and positive values may not be worthwhile. Cook and Lee discovered that 66% of array references could be accomplished in a single indexing operation when the index is shifted by the size of an element in the array. Therefore, an aligned indexing addressing mode would be useful. An analysis of assignment statements disclosed that most of the assignments assigned values to simple variables and had no binary operators. This revealed that simple data movement instructions would be used frequently.

1.2.4.2. Studies Using Architectural-Level Measurements

Lunde [Lun77] used forty-one programs written in five different high-level languages in collecting measurements on the DECsystem10. He collected dynamic measurements by analyzing a trace of a program's execution. He used these measurements to determine the number of registers needed for the machine by inspecting the maximum number of registers being used at any point in the program. He found that, in general, he could generate code that was almost as efficient using only eight of the sixteen registers available on the DECsystem-10. He discovered that 75% of the instructions executed were from less than 11% of the possible instructions. This implies that encoding the commonly executed instructions could be effective to reduce the average size of an instruction. Over 40% of the instructions were moves between a register and memory. Almost 30% of the instructions executed were branches. As a result of analyzing these measurements he suggested a need to improve the calling sequence, memory-to-memory moves, type conversions, and loop control. Lunde also used the trace data to determine the most frequently occurring pairs and triples of sequentially executed instructions to suggest new instructions.

Peuto and Shustek [PeS77] collected dynamic measurements using program tracing from seven programs executed on an IBM 370 and AMDAHL 470. They used a trace program to capture the characteristics of each instruction that was executed. An analysis program was implemented as a coroutine to avoid huge trace files. Use of the trace and analysis programs resulted in programs executing 300 times slower than normal. Peuto and Shustek found that between 50% to 60% of branch destinations were within 128 bytes of the branch instruction. This showed the usefulness of program-counter-relative branch instructions. They found that the average number of bytes of instructions executed between successful branches was less than thirty-two bytes. This justifies the choice of thirty-two bytes for the linesize of the cache on both of the machines. They examined pairs of opcodes to suggest new instructions and possible bottlenecks due to pipeline conflicts. Peuto and Shustek analyzed the dynamic data gathered from the trace program to estimate the execution time of the programs on the two machines. They examined the instruction interactions to determine the penalties due to pipeline conflicts. Using the addresses from operands and instructions, they simulated cache memory to determine the penalties of cache misses. They determined the percentage of execution time required by each instruction by using the number of times that an instruction was executed and the estimated speed of the instruction. Peuto and Shustek found that the most frequently executed instructions were often not the ones which accounted for most of the execution time. This showed that frequency counts alone can be misleading when attempting to determine the influence of a specific instruction. The ability to accurately estimate the execution time of a program can also be used to predict the performance effect of a future change in a machine.

The Computer Family Architecture committee [FuB77] established a set of criteria to measure computer architectures. In the CFA architecture evaluations [FSB77], twelve programs were used to evaluate and rank nine different architectures. The influence of specific architectural features was not evaluated. Most of the test programs were small, usually requiring less than 200 static machine instructions. Each test program was hand-coded in the assembly language of each machine. The measurements used in the evaluation included program size, number of memory transfers, and number of register transfers. The method of obtaining most of these measurements was to simulate code that was generated in an ISP notation [BSG77].

Ditzel [Dit80] performed a static analysis of machine code on the Symbol computer. Using program analysis to collect static measurements of object code generated from five programs written in SPL, he analyzed the instruction frequency, space usage, and jump distances. He found that on the Symbol computer, a stack machine, most of the program space is used by literals and addresses. He discovered that 50% of the jump destinations were within thirty-two words of the jump instruction. He suggested the use of short forms for identifier references, relative jumps, and small constants.

Sweet and Sandman [SwS82] performed a static analysis of Mesa object code. They determined the more frequently occurring types of instruction operands, types of instructions, and pairs and triples of instructions. Using an iterative design method they proposed new instructions and determined the effect of each change. These new instructions were either more compact versions of frequently occurring instructions or combinations of frequently occurring operations. McDaniel [McD82] collected dynamic measurements from two Mesa programs on a Dorado personal computer by microcode instrumentation to measure the frequency of instruction use. He used the measurements for an evaluation that included instruction formatting, the instruction fetch unit, and pipelining requirements. The Mesa studies showed that program measurements are useful for a variety of applications.

Wiecek [Wie82] gathered dynamic measurements of VAX-11 instruction set usage from the execution of six different compilers. She used VAX-11 instruction trace software to trap on each instruction executed to generate the trace data. Programs were then used to analyze instruction frequency, operand specifiers, data memory references, register utilization, instruction sequencing, and branch displacements. The move instruction was the most frequently used instruction and the most frequently used addressing modes were the simpler modes. These modes were register (39.9%), immediate (15.2%), byte displacement (14.2%), and register deferred (7.8%). These results showed that architectures should effectively support simpler instructions and addressing modes. Of the eleven most frequently used instructions, the return and call instructions were the most time consuming. Wiecek also found that there was an average of 3.9 instructions between branches. These observations demonstrate the large influence of calls and branches on the execution of programs.

Cook and Donde [CoD82] used the results from a study of Pascal programs [CoL82] as guidelines in designing the instruction set for MCODE, a stack machine. This machine is designed to execute instructions compiled from programs in StarMod, a distributed programming language based on Modula. The StarMod compiler had already been retargeted to the VAX and the PDP-11. They used a ‘‘set mode’’ instruction to control whether arithmetic operators were performed as real or integer operations. They found that the mode of instructions was changed infrequently. Cook and Donde also encoded more compactly the most frequently used instructions. These modifications resulted in an overall reduction in program size. They measured the sizes of thirty programs compiled on the three machines. The VAX code contained over twice the number of bytes required by the code for MCODE and the PDP-11 code required 50% more space.

Clark and Levy [ClL82] used a hardware monitor on the VAX-11/780 to measure the frequency and execution time of each VAX-11 instruction. Only ten instructions represented over 60% of the instructions that were executed. Simple data moves were the most common type of instruction and branches were the second most frequent. This result again showed that the simpler instructions tend to be heavily used. They found that programs compiled from different languages or applications have the potential for using the instruction set differently.

Patterson and Sequin [PaS82, PaP82] collected static and dynamic measurements from eight Pascal and C programs. Patterson used the results of his study and previous studies in the design of the RISC I machine. Since the results showed that the simpler instructions and addressing modes are the most frequently used, he used a reduced instruction set to make the instruction cycle fast. He determined that the register window approach was feasible since most variable references and arguments to functions were scalars and the average calling depth was not very deep. Since branches occurred very frequently in executed instructions, he used a delayed branch technique to avoid pipeline delays. Patterson used the results of this study and the RISC I and RISC II efforts to suggest new computer design principles [Pat85] that stress fast instruction cycle time, simple decoding, pipelined execution, and compiler technology.

Huck [Huc83] compared four different architectures using program tracing and emulator instrumentation. The most complex of the architectures was found to execute the fewest instructions while the simplest of the architectures had the fastest execution time per instruction. Performance of the architectures was affected by not only differences in the architectures but also other factors such as differences in compilers. Huck then analyzed the effects of compiler optimization techniques. He discovered that fixed-point computation and memory references were reduced while branching instructions were typically unaffected. This showed that as more compiler optimizations are applied, the percentage of control instruction increases.

Wirth [Wir86] compared three architectures based on code density and simplicity of compilation. The architectures were the National Semiconductor 32000, Motorola 68000, and Lilith. The same front end of a Modula-2 compiler was used for all three machines. The same degree of optimization was used in each back end. Since a stack machine does not have directly addressed registers, optimizations such as allocating variables and arguments to registers were not performed. The Lilith, a stack machine designed for executing compiled Modula programs, generated more total instructions, but required fewer total bytes for the compiled code. The Lilith code generator was also found to be smaller than the compilers for the other two machines. This study showed that code generators for simple stack machines would typically be simpler than machines with complex instruction sets and directly addressable registers. This is due to having fewer instructions that can be selected when generating code and fewer optimizations that can be performed. The study also revealed that compilers for stack machines would typically produce smaller executable programs due mostly to arithmetic instructions having implicit operands on the expression stack.

Cook [Coo89] collected static measurements from Lilith machine instructions that were produced by compiling Modula-2 programs obtained from system software. These programs included the operating system, the Modula-2 compiler, and text editors. The compiled software contained over 146,000 instructions. Most of the instructions on the Lilith are represented in a single byte. These short instructions are partly possible since the Lilith is a stack machine which allows instructions to implicitly reference operands. Other instructions that occur frequently are encoded to save space. Cook found that twenty of the 256 instructions represented 50% of the instructions. This observation suggests that the scheme of encoding a specific subset of the instructions in the instruction set can be effective. He discovered that most operations were performed in types defined by the user rather than predefined types. He studied frequently occurring pairs of instructions to suggest new instructions. He also compared the distribution of instructions on the Lilith to the functionally equivalent set of instructions for the Mesa architecture on the Dorado. The Lilith was designed to support Modula-2 and the Dorado was designed to support Mesa. He found that the differences that occurred in the distribution of instructions between the two machines were due mostly to code generation strategies and language usage.

Adams and Zimmerman [AdZ89] collected dynamic measurements using program tracing from the execution of seven programs on an Intel 8086. Only twenty-five instructions, or 25% of the total 8086 instruction set, were responsible for about 91% of all the instructions executed. This shows that only a fraction of the available instructions are used very frequently. Register to register moves appeared very frequently. This suggests that the specialized use of registers on the 8086 resulted in many registers being shuffled around. Adams and Zimmerman compared their results to previous studies on the VAX-11 [Wie82] and found many of the most frequently used instructions for both machines were the same. This occurred despite the VAX-11 having a much more complex instruction set.

CHAPTER 2

METHOD FOR GATHERING DATA

To evaluate an instruction set effectively, one must collect measurements from the program's execution that can be used to evaluate the influence of specific features of the instruction set. As described previously, program instrumentation captures this data for subsequent analysis with little overhead. One way to accomplish program instrumentation efficiently is to modify the back end of the compiler to store the characteristics of the instructions to be executed and to produce code which will count the number of times that each instruction is executed. These modifications have been implemented in a portable optimizer called *vpo* (Very Portable Optimizer) [BeD88] and are described in subsequent sections. The environment that comprises the modified compiler and programs used to produce reports is called *ease* (Environment for Architecture Study and Experimentation) [DaW90a] and is illustrated in Figure 1.

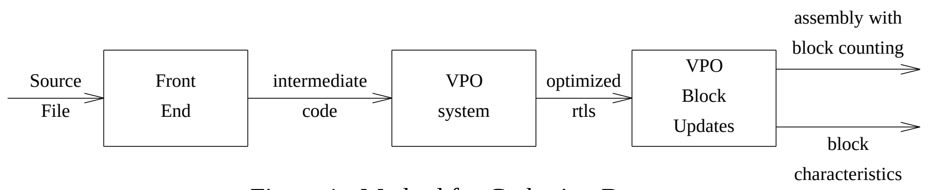


Figure 1: Method for Gathering Data

2.1. Instruction Characteristics

The optimizer, *vpo*, replaces the traditional code generator used in many compilers and is retargeted by supplying a description of the target machine, a *yacc* grammar. Instructions are stored as character strings using ISP-like register transfers [BeN71]. For example, the register transfer list (RTL)

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

typically represents a register-to-register add on a machine. Though each RTL is machine-specific, the form allows optimization of RTLs to occur in a machine-independent fashion. One optimization performed by the compiler is instruction selection. *Vpo* links instructions that are candidates to be replaced by a more complex instruction that performs the same function. This is accomplished by symbolically simulating pairs and triples of linked instructions to produce a combined effect. The machine description is used as a recognizer to determine if the combined effect is a legal instruction.

The first modification to *vpo* necessary to collect measurements is to have the optimizer 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 by *vpo*. The semantic checks are modified to store these characteristics with the RTL 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 2.

```
/*
 * call - written by compiler writer to check semantics of call instruction
 */
void call(i1)
struct sem_rec *i1;
{
   /* if generating assembly code */
   if (dassem)
      printf("\tjbsr\t%s\n", i1->sem.call.addr->asmb);

   /* else if no semantic errors and measurements are to be collected */
   else if (!erflag && swm)
      stinstinfo(JSBRI, i1);
}
```

Figure 2: Storing Instruction Information

2.2. Frequency Counters

The second modification is to have the optimizer generate code after all optimizations have been performed to count the number of times each instruction is executed. 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 are denoted as *execution classes*. Thus, the code that the optimizer generates to count the number of times that each instruction in an execution class is executed is inserted in only one basic block in the execution class.

An example of inserting frequency counters is given in Figures 3—6. Figure 3 shows a C function. Figure 4 gives the VAX-11 assembly code that would normally be produced by *vpo* for that function.

```
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 3: 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 4: Vax Assembly Code for Function in Figure 3

Figure 5 contains 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}).

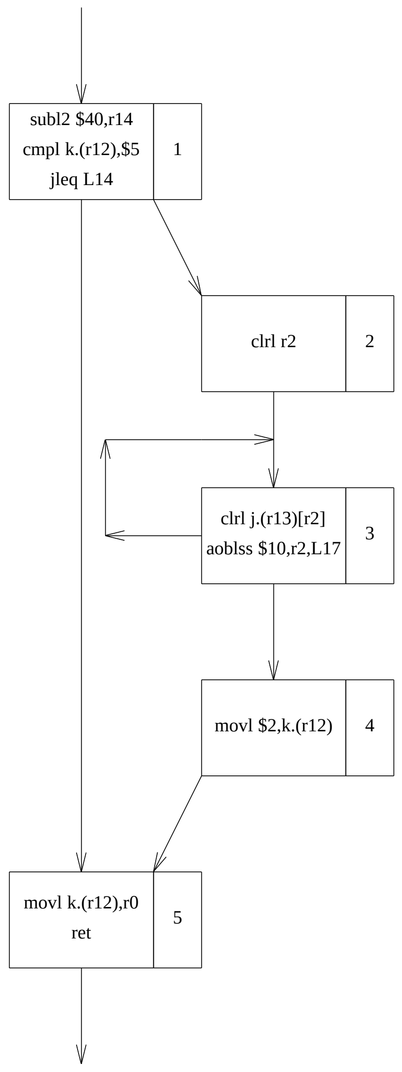


Figure 5: Assembly Code of Figure 4 in Basic Blocks

Figure 6 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

```
.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 6: Vax Assembly Code with Frequency Counters

is always a successor to the dominator blocks. This determines 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*. The algorithms for computing both the dominators of blocks and the blocks in a loop are given in ASU86. Figure 7 presents the algorithm for determining the always successor information.

```
for each block n
   set AS(n) to n
while changes to any AS(n)
   for each block n
      for each block b in AS(n)
         for each predecessor p of block b
            if all successors of p are in AS(n)
               add AS(p) to AS(n)
```

Figure 7: Calculating Always Successor Information for each Block

Figure 8 shows the dominators (DOM), always successors (AS), and execution classes (EC) for the set of blocks in Figure 5.

$$EC = (DOM \cap AS) - DIFFLOOPS$$

```
DOM(1) = {1}        AS(1) = {1}          EC(1) = {1}
DOM(2) = {1,2}      AS(2) = {2}          EC(2) = {2}
DOM(3) = {1,2,3}    AS(3) = {2,3}        EC(3) = {3}
DOM(4) = {1,2,3,4}  AS(4) = {2,3,4}      EC(4) = {2,4}
DOM(5) = {1,5}      AS(5) = {1,2,3,4,5}  EC(5) = {1,5}
```

Figure 8: Example of Execution Classes

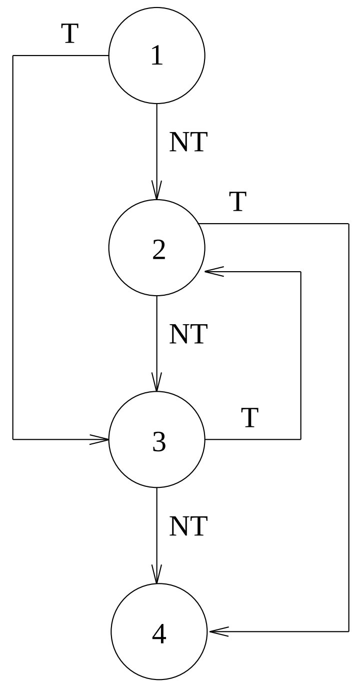
2.3. Other Modifications

Another measurement needed is instruction size information that is calculated using the assembler on each machine. Labels are inserted before and after each basic block. The difference between the label at the end of the block and the label at the beginning of the block represents the number of bytes of the instructions in the basic block. The size of each basic block in the execution class are then added together. Figure 9 shows the VAX-11 assembly code from Figure 6 with labels inserted to determine the size of each execution class. Each element of the `sizes` array contains the number of bytes in an execution class.

Some types of measurements require additional code to be generated. For instance, the number of times that conditional branches are taken can be accurately 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. At first glance, it may appear that one can use the frequency counts from the execution classes to determine this information. With frequency counts alone, however, one cannot produce an accurate measurement with unstructured code. For example, Figure 10 shows two execution paths that may be taken thru a set of basic blocks that would produce identical frequency counts. In the first path three conditional branches are not taken, while in the second the three conditional branches are taken.

```
.text
.globl _foo
_foo:
.word   0x0
.set    k.,4
.set    j.,-40
        incl     _test_counts
LS1_1:
        subl2    $40,r14
        cmpl     k.(r12),$5
        jleq     L14
LE1_1:
        incl     (_test_counts + 4)
LS1_2:
        clrl     r2
LE1_2:
L17:
        incl     (_test_counts + 8)
LS1_3:
        clrl     j.(r13)[r2]
        aoblss   $10,r2,L17
LE1_3:
LS1_4:
        movl     $2,k.(r12)
LE1_4:
L14:
LS1_5:
        movl     k.(r12),r0
        ret
LE1_5:
.data
.globl  _test_sizes
_test_sizes:
.long   LE1_1-LS1_1+LE1_5-LS1_5
.long   LE1_2-LS1_2+LE1_4-LS1_4
.long   LE1_3-LS1_3
```

Figure 9: Vax Assembly Code with Labels for Size Information



Path1: 1 => 2 => 3 => 4

Path2: 1 => 3 => 2 => 4

Figure 10: Measuring When Conditional Branches Are Taken

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.

In the past, some instructions in a *vpo* back-end were generated by the code expander in assembly language if optimizations could not affect it. Every type of instruction that is executed must now be represented as an RTL recognizable by the machine description, however, if its execution characteristics are to be collected.

Depending upon the type of measurements required, special cases may be necessary. For instance, the VAX-11 `movc` instruction moves a variable number of bytes of memory depending upon a value in an operand. To be able to accurately count the number of memory references made by the execution of the instruction, the value of the operand is stored with the other characteristics of the instruction. These special cases only occur when the compiler writer is unable to accurately represent the effects an instruction in the machine description.

2.4. Processing the Collected Data

The data that is collected is 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 a 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 instruction characteristics can also be used to produce the static measurements. Figure 11 shows how both static and dynamic measurements can be obtained.

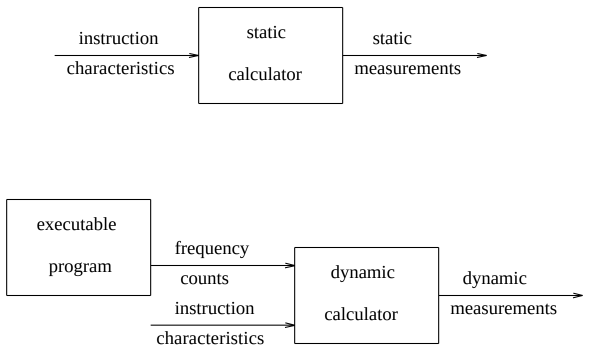


Figure 11: Producing Reports

If the basic block structures for a program are the same on different machines, then the frequency counts of the execution classes should also be identical. The only reason for executing the program on a different machine in this situation is to ensure that the code generated for the program executes correctly. The basic block structures for a program on different machines, however, may often vary. Code generation strategies, such as either returning from several points in a function or always branching to one location to return, can affect the block structure. Special instructions, such as the `movc` instruction on the VAX-11 to accomplish structure assignments that would otherwise require a loop, can also change the number and structure of basic blocks. Since there is little execution overhead to obtain the frequency counts, it is simpler to execute the program on each machine instead of trying to first determine if the basic block structures are identical.

2.5. Discussion

The *vpo* optimizer for ten different machines, already ported and tested, was modified to collect measurements as specified in the previous sections. It typically took three or four hours for an experienced person to make the machine-dependent modifications for the compiler on each machine. For the resulting compiled programs, there is little overhead for collecting data to produce the proposed 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. When execution classes were not used and counters were placed in each basic block, `whetstone` would have required eighty-four counters and 9% more execution time.

The generation of reports from the measurements is also mostly machine-independent. Most of the code from a 500 line C program that produces several different reports has remained unchanged when implemented on the different machines. These reports gather data on the following:

1. instruction type distribution
2. addressing mode distribution
3. memory reference size distribution
4. register usage
5. condition code usage
6. conditional branches taken
7. data type distribution

CHAPTER 3

ARCHITECTURAL STUDY

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 in length.

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. Johnson's Portable C Compiler (*pcc*) [Joh79] was retargeted to each machine in Patterson's study [PaP82]. Thus, Patterson claimed that the different compilers in his study used the same compiler technology. The quality of the code produced by each *pcc* compiler, however, depends on the skill of the compiler writer when constructing tables for code generation and the patterns for peephole optimization.

The methods used to accomplish several of the past architectural studies made it difficult to capture certain kinds of information and perform various experiments. Data was collected from machine instructions in many studies without modifying the compiler. The methods used included simulation [BSG77], trace software [Lun77, PeS77, Wie82], and hardware monitors [ClL82]. Capturing specific types of measurements, such as the number of registers used only as temporaries, is not possible with these methods. Furthermore, determining the usefulness of proposed architectural features is difficult without the ability to modify the compiler and obtain information showing how frequently the proposed features are used.

Ease has eliminated problems that have limited some past architectural studies. Using *ease* to collect data, one can use a number of realistic programs and collect the data in a timely fashion. For example, on the VAX-11/8600, measurements were collected from the execution of almost 100 million instructions in less than ten minutes. Properties of *vpo*, the optimizer used in *ease*, eliminate several problems. Since code to perform instruction selection and most optimizations is 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 [DaW89]. For example, tables are constructed by the implementor of a *pcc* compiler to recognize the different instances of trees from which an instruction can be generated. Instructions for *vpo*, however, only have to be represented correctly by the machine description. Retargeting the compiler to a new machine only requires expanding the intermediate language statements to RTLs and describing the architecture. Ad hoc case analysis is unnecessary. Thus, the programs compiled for each of the architectures receive the same degree of optimization. Because an effort was made to separate the machine-independent code from the machine-dependent code to facilitate the retargeting of *vpo*, changing the compiler to implement proposed architectural changes such as reducing the number of registers available, changing the calling sequence, or eliminating an instruction or addressing mode is relatively easy to accomplish in the *vpo* compiler system. Even adding a new instruction or additional registers usually can be done easily since one RTL can be translated to one or more machine language instructions.

The following sections describe a study of several architectures that involved collecting measurements from the execution of the same set of test programs on each machine. First, the type of measurements extracted by *ease* from each architecture is given. Next, the characteristics of each of the architectures in the study is described. The set of test programs used in the study is then specified. Finally, the measurements obtained from the architectures are analyzed.

3.1. Types of Measurements

There are several kinds of data that can be collected to aid in the design, modification, or evaluation of an instruction set. The following is a list of some of the data that has been collected both statically and dynamically using the modified version of *vpo*.

1. total number of instructions
2. number of each type of instruction
3. instruction size information
4. number of each type of addressing mode used
5. number of memory references
6. number of each different size of memory reference
7. number of each different memory reference addressing mode
8. condition code usage
9. distribution of size of data for each instruction
10. distribution of addressing modes for each field in each type of instruction
11. number of each type of register saved and restored
12. register usage

Dynamic data has been collected on the number of conditional branches taken and not taken and the number of instructions executed between branches.

These measurements can be used in a variety of ways. The frequency information can be used to evaluate the usefulness of specific types of instructions, addressing modes, or other architectural features. It can also be used to help determine how the instructions should best be encoded. The total number of instructions and memory references can help to evaluate the effect of changing a specific feature of the architecture.

3.2. Architectures Studied

The architectural study includes the following architectures:

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

These architectures differ in a number of ways including the number and complexity of instructions and addressing modes, the number and use of registers, and degree of orthogonality of the instructions and addressing modes. The following subsections describe the code that *vpo* generates and features of the architectures for each machine in the study. The definition of calling sequence conventions, often influenced by the available instructions or mechanisms in the architecture, can also affect the code that is generated. The calling conventions as defined on the machines were used since the source files of the run-time library on most of the machines in the study were not accessible.

3.2.1. VAX-11

The VAX-11/780 has an 8K byte cache used to store data and instructions. There are twelve user-allocable registers. There are six scratch registers and six non-scratch registers. Doubles are referenced in pairs of registers.

The instruction set for the VAX-11 is considered complex and very orthogonal. The *vpo* compiler for the VAX-11 can generate fifty-three different instructions and twenty-two different addressing modes for instruction operands without regard to datatype. Most arithmetic instructions allow a two-address (one source and a source that is the same as the destination) or three-address format (two sources and a destination). The supported data types are:

1. BYTE	(8 bits)
2. WORD	(16 bits)
3. LONG	(32 bits)
4. FLOAT	(32 bits)
5. DOUBLE	(64 bits)
6. QUAD	(64 bits)

A callee-save calling sequence is used (the function that is called has the responsibility of saving and restoring registers that are used).

3.2.2. Harris HCX-9

The HCX has an 16K byte data cache as well as an instruction cache that can store up to 4K decoded instructions. Of the thirteen user-allocable registers, two are scratch registers and eleven are non-scratch registers. Doubles are referenced in pairs of registers. There is also one floating-point accumulator which is used by all floating-point operations. The clock period is 100 nsecs and there is a three stage pipeline.

The fixed-point portion of the instruction set for the HCX is very similar to the VAX-11. The compiler for the HCX can generate fifty-four different instructions and twenty-two different addressing modes for instruction operands without regard to datatype. Most integer arithmetic instructions allow a two-address or three-address format. Floating point operations are accomplished in a one-address format (implicit accumulator). The supported data types are:

1. BYTE	(8 bits)
2. WORD	(16 bits)
3. LONG	(32 bits)
4. FLOAT	(32 bits)
5. DOUBLE	(64 bits)
6. QUAD	(64 bits)

A callee-save calling sequence is used.

3.2.3. AT&T 3B15

The AT&T 3B15 has twelve allocable registers, nine general-purpose (fixed-point) and three floating-point. Three of the general-purpose registers are scratch and all three of the floating-point registers are scratch. Each floating-point register can contain a single or double-precision value.

The instruction set for the 3B15 is very similar to the VAX. The compiler implemented for the 3B15 can generate fifty-five different instructions and eleven different addressing modes for instruction operands without regard to datatype. As in the VAX-11, most arithmetic instructions allow a two-address or three-address format. Unlike the VAX-11, there are no indexed addressing modes. The supported data types are:

1. BYTE	(8 bits)
2. HALF	(16 bits)
3. WORD	(32 bits)
4. FLOAT	(32 bits)
5. DOUBLE	(64 bits)

A callee-save calling sequence is used.

AT&T has designed several machines based on the 3B architecture. There are also a few differences in the set of instructions for some of these machines in this family. AT&T defined a set of machine-independent instructions, the IS25 Instruction Set, to allow assembly programs to be portable between each of the machines. Only machine-independent instructions within the 3B family were generated by *vpo* due to portability considerations and lack of documentation on machine-dependent instructions. In some cases machine-independent instructions are expanded by the assembler into several machine-dependent instructions.

3.2.4. Motorola 68020/68881

There are different versions of the 68020 and 68881 floating-point coprocessor for the SUN-3/60 available from Motorola. For these experiments the 68020 has a 256 byte on-chip instruction cache. The twenty-two user-allocable registers are separated into three classes to save bits in the instruction format. There are eight data (integer) registers, eight floating-point registers, and six address registers. Each of these classes has two scratch registers. Floating-point registers can contain either single or double-precision values.

The instruction set for the 68020 is moderately complex and unorthogonal. For example, the only immediate values allowed for the shift instruction is between 1 and 8. Many instructions do not allow operands to reference memory or specific addressing modes. These types of restrictions are to allow tighter encoding of instructions and less use of area on the chip. The *vpo* compiler for the 68020 can generate forty-five different instructions and nineteen different addressing modes for instruction operands without regard to datatype. Arithmetic instructions typically have a two-address format (one source and a source that is the same as the destination). The supported data types are:

1. BYTE	(8 bits)
2. WORD	(16 bits)
3. LONG	(32 bits)
4. FLOAT	(32 bits)
5. DOUBLE	(64 bits)

A callee-save calling sequence is used.

3.2.5. National Semiconductor 32016

The instruction set for the 32016 is considered moderately complex and very orthogonal. The *vpo* compiler for the 32016 can generate forty-seven different instructions and seventeen different addressing modes for instruction operands without regard to datatype. Most arithmetic instructions allow a two-address format. The supported data types are:

1.	BYTE	(8 bits)
2.	WORD	(16 bits)
3.	LONG	(32 bits)
4.	FLOAT	(32 bits)
5.	DOUBLE	(64 bits)

There are sixteen user-allocable registers divided into two classes, eight integer registers (three scratch) and eight floating-point registers. A callee-save calling sequence is used for integer registers. A caller-save calling sequence, however, is used for floating-point registers (the calling function has the responsibility of saving and restoring registers). Doubles are referenced in even-odd pairs of registers.

3.2.6. Intel 80386/80387

The *vpo* optimizer was also ported to the 80386 and 80387 floating-point coprocessor. The fetching and execution of an instruction is implemented in eight different units that operate in parallel. The 80386 integer registers overlap for different datatypes. There are six long (32 bit) registers (three scratch), four word (16 bit) registers, and eight byte (8 bit) registers. In the 80387 there are eight floating-point registers implemented as a stack.

The instruction set for the 80386 is considered unorthogonal. Several operations are required to be performed in specific registers. Typically an instruction can only reference memory with at most one operand. The compiler for the 80386 can generate forty-eight different instructions and fourteen different addressing modes for instruction operands without regard to datatype. Most arithmetic instructions allow a two-address format. The supported data types are:

1. BYTE	(8 bits)
2. WORD	(16 bits)
3. LONG	(32 bits)
4. FLOAT	(32 bits)
5. DOUBLE	(64 bits)

A callee-save calling sequence is used for integer registers and a caller-save calling sequence is used for floating-point registers that are alive across calls.

3.2.7. Concurrent 3230

The compiler implemented for the Concurrent 3230 can generate 101 different instructions and eleven different addressing modes. The large number of different types of instructions was due to mnemonic instruction opcodes being used to depict the type and size of the operands. For instance, the following instructions represent addition operations:

1. a	- add a value from memory to a register
2. ar	- add a register to a register
3. ai	- add a constant to a register
4. ais	- add a 4-bit constant to a register
5. ah	- add a halfword value from memory to a register
6. am	- add a register to memory
7. ahm	- add a register to a halfword of memory

Arithmetic operations are accomplished in a two-address format. The supported data types are:

1. LONG	(32 bits)
2. FLOAT	(32 bits)
3. DOUBLE	(64 bits)

Whenever a BYTE (8 bits) or WORD (16 bits) is loaded from memory it is converted to a LONG type.

The Concurrent 3230 has sixteen general-purpose registers, eight single-precision floating-point registers, and eight double-precision floating-point registers. A caller-save calling sequence is used.

3.2.8. IBM RT/PC

The IBM RT/PC system processor and floating-point accelerator have a reduced set of operations and addressing modes. The compiler implemented for the IBM RT/PC can generate sixty-two different instructions and eight different addressing modes. The instruction opcodes are used to depict the type and size of the operands. For instance, the following instructions represent addition operations:

1. a	- add a register to a register
2. ai	- add a 16-bit constant to a register
3. ais	- add a 4-bit constant to a register

The supported data types are:

1. LONG	(32 bits)
2. FLOAT	(32 bits)
3. DOUBLE	(64 bits)

Whenever a BYTE (8 bits) or WORD (16 bits) is loaded from memory it is converted to a LONG type. Memory is only accessed by load and store instructions. All branches have the option of allowing the instruction immediately following the branch to be executed while the instruction at the branch address is being fetched. Multiplication by a register is expanded into sixteen multiplication step instructions. Multiplication by a constant is expanded into a sequence of shift, add, and subtract instructions.

The IBM RT/PC system processor has sixteen general-purpose registers. Only twelve of the sixteen registers can be allocated for general use by the compiler. Four of these registers are scratch and eight are non-scratch.

The system uses an unusual calling sequence. Only one register is used for the functions of both a stack and a frame pointer. The maximum number of longwords passed as arguments to other functions is required to be known to determine the offsets of local variables and temporaries. Up to four longwords are passed in the four scratch registers. If more than four longwords are required then they are passed on the stack. A double-precision or structure argument may have to be partially passed both in registers and the stack.

Immediate operands are limited to sixteen bits for most instructions. Constants requiring more than sixteen bits must either be constructed or loaded from memory in a constant pool. The addresses of any globals that are referenced in the function are also kept in a constant pool. A general-purpose register is dedicated for use as a constant pool pointer.

The floating-point accelerator has sixteen user-allocable registers. Only twelve of the sixteen registers can be allocated for general use by the compiler. Doubles are referenced in even-odd pairs of floating-point registers. All of these registers are scratch. All floating-point instructions are initiated by the system processor by load and store instructions using memory-mapped I/O. A floating-point value that is transferred between a floating-point register or memory must pass through a general-purpose register.

3.2.9. Intergraph Clipper

The Intergraph Clipper is considered a reduced instruction set machine. The compiler implemented for the Clipper can generate forty-nine different instructions and nine different addressing modes without regard to datatype. The supported data types are:

1. LONG	(32 bits)
2. FLOAT	(32 bits)
3. DOUBLE	(64 bits)

There are many similarities between the IBM RT/PC and the Clipper. BYTE (8 bits) or WORD (16 bits) values loaded from memory are converted to a LONG type. Memory is only referenced by load or store instructions. Unlike the RT, there are no delayed branches and floating-point instructions are encoded directly into the instruction set.

The Clipper has fifteen general-purpose registers and eight registers that can contain single-precision or double-precision values. Six of the general-purpose registers and four of the floating-point registers are scratch.

Like the RT, the Clipper has no frame or argument pointer and passes some of the arguments through registers. The Clipper allows arguments to be passed through up to two general-purpose or floating-point registers depending on the argument type. Structure arguments are always passed on the stack.

3.2.10. SUN SPARC

The SUN SPARC has a reduced set of operations and addressing modes. The compiler implemented for the SPARC can generate forty-one different instructions and seven different addressing modes without regard to datatype. The supported data types are:

1. LONG	(32 bits)
2. FLOAT	(32 bits)
3. DOUBLE	(64 bits)

As in the other RISC machines in the study, the IBM RT/PC and the Clipper, the SPARC only allows references to memory by load or store instructions (where references to BYTEs (8 bits) or WORDs (16 bits) can be accomplished). The delayed branch for the SPARC has two differences from the delayed branch for the IBM RT/PC. Unlike the SPARC, the RT allows the option of not executing the instruction following the branch for each type of branch instruction. The SPARC instead has an option to not execute the instruction following a conditional branch when the conditional branch is not taken.

The SPARC is the only machine in this study to use register windows. Each of the eight windows has access to four different sets of eight registers listed below.

1. global
2. output
3. local
4. input

There are sixteen unique registers in each window where the output registers of one window overlap the input registers of the next window. This overlap allows arguments to passed through the scratch output registers and be received in the non-scratch input registers. The eight non-scratch locals registers are unique to each window while each window can access the same eight scratch global registers. Twenty-seven of the registers that were described are allocable. Two input and two output registers are used for function linkage and one of the global registers always contains a value of zero.

3.3. Test Set

Using *ease*, realistic programs can be compiled and executed without excessive delay. The test set used in this study consists of nineteen C programs, forty-five files, and over 23000 lines of source code. For most machines, when the test set is run, over 100 million instructions are executed. At this time only a C front end, *vpcc* [Wat86], has been used with the *vpo* optimizer. Programs written in other high-level languages would provide additional useful information. Since many UNIX systems mainly execute compiled C programs, a set of C programs can still be fairly representative. Table 1 summarizes the test set used in this study. Both the number of lines of code and intermediate code operations [DaW88] are given. The number of lines of code of two identical programs may vary depending upon the comments and style used by the programmers. Since the number of intermediate code operations is not affected by these factors, it may be a more accurate measure for judging the relative sizes of the programs.

Class	Name	Description or Emphasis	Lines of Code	Inter Opers
Unix System Utilities	cal	Calendar Generator	204	1341
	cb	C Program Beautifier	359	2645
	compact	Huffman Coding File Compression	420	3593
	diff	Differences between Files	972	8868
	grep	Search for Pattern	532	3201
	nroff	Text Formatting Utility	6948	34627
	od	Octal Dump	894	5161
	sed	Stream Editor	1692	13571
	sort	Sort or Merge Files	914	6564
	spline	Interpolate Smooth Curve	338	2472
	tr	Translate Characters	133	1374
	wc	Word Count	104	638
Benchmark Programs	dhrystone	Synthetic Benchmark Program	731	1421
	matmult	Multidimen Arrays and Simple Arith	113	605
	puzzle	Recursion and Array Indexing	235	2586
	sieve	Simple Iteration and Boolean Arrays	54	362
	whetstone	Arithmetic Operations	327	2447
User Code	mincost	VLSI Circuit Partitioning	494	3715
	vpcc	Very Portable C Compiler	8417	62038

Table 1: Test Set

3.4. Analysis of Measurements

One method of comparing different machines is to measure the time it takes to execute programs. Execution times are typically easy to obtain and represent one number for evaluation. Execution times, however, may also provide imprecise information. For instance, ten executions of a program can result in ten different execution times. The elapsed time can change dramatically since the load on the machine can vary with each execution. Most operating systems provide the feature of estimating the portion of the elapsed time used by the executing user program. The user time, though more accurate than the elapsed time, can still vary. The estimation of time spent by each program is typically accomplished by examining which process is currently executing at periodic intervals. In order not to cause much overhead, these intervals are typically greater than the time required for switching processes. This variance can be greater than the effect from a change in the compiler or architectural implementation. Therefore, the ability to obtain more accurate measurements would be desirable.

Comparing execution times can also be misleading when attempting to compare architectures since there are many other facets of a machine that can affect execution speed. For instance, Figure 12 shows the execution rate when executing the benchmark *dhrystone* on each of the architectures.

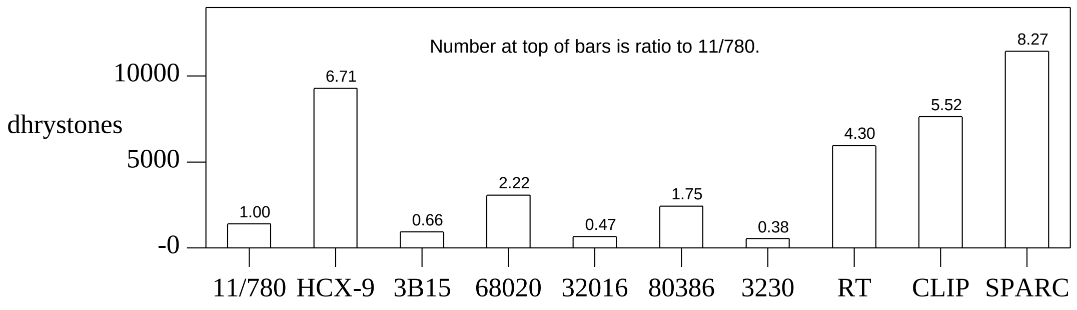


Figure 12: Number of Dhrystones that each Machine Executes

The VAX-11/780 required almost seven times the execution time of the HCX-9 to execute the benchmark *dhrystone*. There are no floating-point operations required in *dhrystone*. The set of fixed-point instructions on the VAX is very similar to the set of fixed-point instructions on the HCX. Other features of the HCX, such as a large instruction cache and faster clock rate, are probably responsible for the most of the difference in execution times between the two machines. Therefore, it would be more appropriate to measure the frequency of use of a specific architectural feature in order to determine its influence, rather that just measuring the execution time of the machine.

3.4.1. Analysis of Dynamic Measurements

Various aspects of the dynamic measurements of the architectures in the study were analyzed. The insights that were obtained from the analysis is given in the following sections.

3.4.1.1. Instruction Path Length

The instruction path length is the total number of instructions executed. In general, the more complicated the instruction set, the shorter the path length became as shown in Figure 13. Note that each floating-point instruction in the RT floating-point accelerator is initiated by two fixed-point instructions using memory-mapped I/O. This scheme increased the instruction path length for the RT by 1.6%. The RT and SPARC were the only machines that used a delayed branch. Figure 13 shows the instruction path length of each machine by solid lines. It also shows by dashed lines the adjusted instruction path length of each machine assuming there is an implied no-operation instruction after each transfer of control that is not a delayed branch.

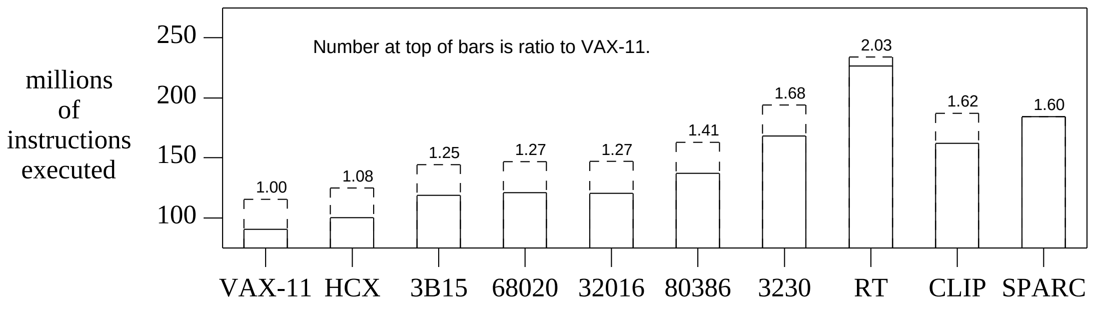


Figure 13: Instruction Path Length

3.4.1.2. Instruction Path Size

The instruction path size is the total number of bytes of instructions that were executed. The path size divided by the total number of instructions executed (instruction path length) determines the average number of bytes per instruction, as depicted in Figure 14 and Table 2. Note that since machine-independent instructions for the 3B15 family were used, some of these instructions are expanded into several machine-dependent instructions.

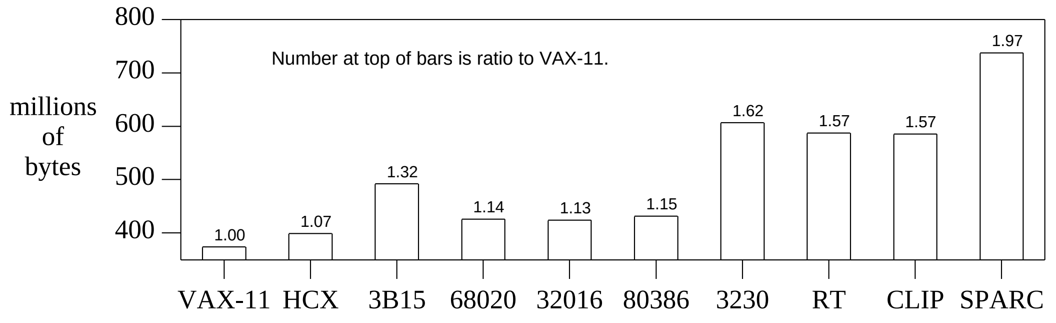


Figure 14: Instruction Path Size

Machine	Bytes per Instruction
VAX-11	4.14
HCX	3.98
3B15	4.15
68020	3.52
32016	3.52
80386	3.15
3230	3.61
RT	2.59
CLIPPER	3.61
SPARC	4.00

Table 2: Average Bytes per Instruction Executed

Except for the 3230, the Clipper, and the SPARC, machines with a smaller number of executed instructions had both a greater number of bytes per instruction and a smaller number of total executed instruction bytes. The number of instructions executed by the 3230, the Clipper, and the SPARC was only 74%, 72%, and 81% respectively of the number of instructions executed by the RT. Yet the 3230 instruction path size was over 3% larger than the instruction path size for the RT and the Clipper instruction path size was less than 1% smaller. This suggests that the encoding of instructions for the 3230 and the Clipper could be improved. The SPARC instruction path size was over 26% larger than the instruction path size of the RT. Since there was no attempt at encoding instructions for the SPARC, every instruction is four bytes in length.

3.4.1.3. Instruction Type Distribution

Instructions were arbitrarily divided into 4 classes.

DATA MOVEMENT
ARITH/LOGICAL
CONTROL
CALL SEQUENCE

These classes overlap somewhat. For instance, some ARITH/LOGICAL and DATA MOVEMENT instructions set condition codes that are used for control purposes (like a compare instruction). The percentages of instructions in each class for each machine are summarized in Table 3. The DATA MOVEMENT instructions for the 68020, 32016, 80386, 3230, and RT were the most frequently used class. The ARITH/LOGICAL and CONTROL classes were used more frequently than the DATA MOVEMENT class for the VAX-11, HCX, and 3B15. This occurred for these three machines since they allow many complicated instructions (for instance, three-address instructions) and are orthogonal with respect to referencing registers or memory. Machines with a more complicated instruction set had a larger percentage of CONTROL type instructions. The number of CONTROL type instructions varied much less between machines than DATA MOVEMENT and ARITH/LOGICAL instructions. A similar number of CONTROL instructions on each machine shows that there has been little emphasis on creating more complicated instructions used for transfer of control.

Machine	Per Cent DATA MOVEMENT	Per Cent ARITH/LOGICAL	Per Cent CONTROL	Per Cent CALL SEQUENCE
VAX-11	27.75	28.39	37.93	5.92
HCX	28.34	32.01	34.22	5.43
3B15	27.85	36.92	30.64	4.59
68020	42.74	18.46	28.73	10.07
32016	45.14	11.65	32.42	10.79
80386	49.43	18.17	26.63	5.77
3230	55.22	20.03	19.61	5.13
RT	49.06	29.72	17.48	3.74
CLIPPER	50.54	22.14	22.38	4.95
SPARC	43.52	35.00	16.56	4.91

Table 3: Instruction Class Distribution

3.4.1.4. Addressing Mode Distribution

The simpler addressing modes were consistently used most frequently on all machines. Table 4 shows the percentage of referencing registers, immediate values, and labels for the seven machines.

Machine	Per Cent of Total Operand References
VAX-11	69.16
HCX	74.87
3B15	76.09
68020	72.87
32016	69.63
80386	69.07
3230	72.80
RT	71.35
CLIPPER	83.15
SPARC	88.55

Table 4: Registers, Immediates, and Labels as a Per Cent of Total Operand References

Displacement from a register was also common due to the frequent reference of locals and arguments.

3.4.1.5. Memory References

A majority of memory references were consistently LONGs on all the machines despite having several character and floating-point intensive programs. The VAX-11 and Clipper had fewer LONG memory references than the other machines due to use of the `movc` instruction that was counted as moving blocks of memory as characters (bytes). The RT required that DOUBLE values be loaded and stored as two FLOAT memory references. The results of memory references by size is given in Table 5.

Machine	Per Cent BYTE	Per Cent WORD	Per Cent LONG	Per Cent FLOAT	Per Cent DOUBLE
VAX-11	10.83	1.16	86.28	1.65	0.08
HCX	5.90	1.16	90.97	1.87	0.09
3B15	6.04	1.19	89.46	3.07	0.25
68020	6.02	1.31	89.78	2.20	0.70
32016	5.28	1.28	90.23	3.04	0.17
80386	7.03	1.10	89.04	2.71	0.13
3230	5.19	0.94	92.58	1.16	0.14
RT	4.71	0.93	92.71	1.65	0.00
CLIPPER	14.20	1.65	81.58	2.26	0.31
SPARC	5.73	1.45	89.03	3.53	0.26

Table 5: Memory Reference Size Distribution

The most common addressing mode used for memory references was displacement (referencing locals and arguments). Direct addressing modes were also used to frequently reference memory (referencing globals). Typically about three tenths of all operand references were to memory. The exceptions were the Clipper, RT, and SPARC, which passed up to two, four, and six arguments respectively to functions in registers. This reduced the number of references to memory since there were fewer operands that were pushed on or referenced from the run-time stack. Also since the three machines only referenced memory through loads and stores, there was a greater percentage of register references. This is shown in Table 6.

Machine	Per Cent with Memory Reference
VAX-11	27.35
HCX	28.06
3B15	29.37
68020	26.14
32016	28.77
80386	28.74
3230	29.63
RT	8.57
CLIPPER	18.10
SPARC	16.33

Table 6: Per Cent of Operand References to Memory

The number of times memory was referenced by each machine is shown in Figure 15. 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. Memory references for function linkage on some machines are not required for other machines. For instance, the 32016 adjusts the stack pointer by an `adjsp` instruction after each call. The additional memory references in the VAX-11 over the HCX was mostly due to *ease* treating the `movc` instruction as only referencing bytes. Over 3.9% of the memory references in the RT were due to initiating the floating-point instructions by two fixed-point instructions using memory-mapped I/O. Global addresses could not be encoded directly in the format for the RT instructions. Each global address had to be loaded from a constant pool. Loads of global addresses accounted for approximately 12% of the memory references on the RT. These methods for implementing floating-point operations and global addresses allowed the average number of bytes per instruction on the RT to be reduced. Since the SPARC used register windows, it had a very small number of memory references for saving and restoring allocable registers and function linkage. The SPARC had very few FLOAT and DOUBLE variables allocated to registers since all thirty-two of the floating-point registers were scratch. FLOAT and DOUBLE variables also occurred infrequently. Since the SPARC had the most allocable general-purpose registers, more variables were allocated to registers on the

SPARC than any other machine in the study. This resulted in the SPARC having the fewest number of memory references. The Clipper had the next smallest number of memory references due to a number of factors which includes having many allocable general-purpose registers, passing some arguments through registers, and not having a frame or argument pointer.

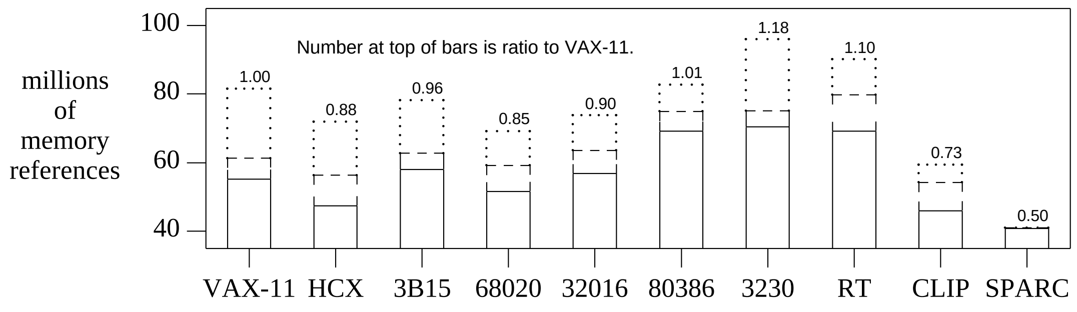


Figure 15: Number of Memory References

The number of bytes transferred to and from memory may be a more meaningful measurement. Figure 16 shows similar information as in Figure 15 except that the number of bytes transferred is given.

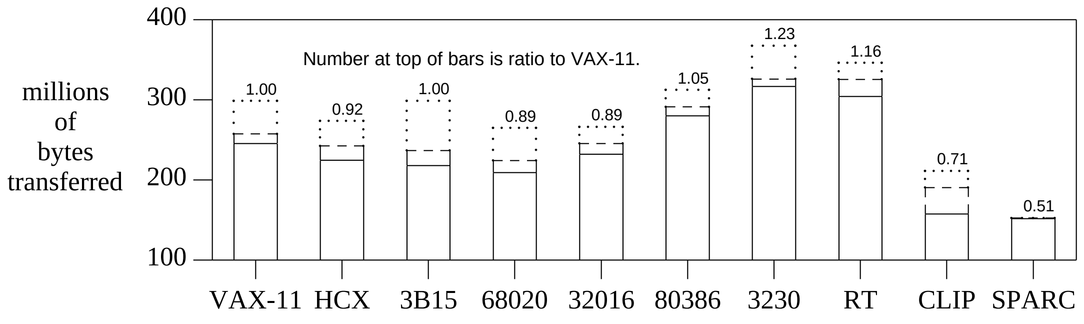


Figure 16: Number of Bytes Transferred to and from Memory

3.4.1.6. Saves and Restores

All the machines except for the 3230 used a callee-save calling sequence for fixed-point registers. The number of saves and restores varied greatly as shown in Table 7. The 3230 value was calculated by dividing the total number of registers saved before each call by the total number of call instructions executed. The caller-save method used resulted in a smaller number of registers saved and restored since fewer variables were allocated to registers. The reduced instruction set of the RT resulted in more registers being used. This shows that a calling sequence convention can have a significant impact on the total number of memory references. The SPARC value was calculated by determining the number of spills and reloads of registers when the number of available windows was exceeded. The small number of saves and restores for the SPARC demonstrates that register windows can effectively reduce the number of memory references for saving and restoring the state of functions between calls.

Machine	Number Per Call	Per Cent of Total Memory References
VAX-11	1.19	9.89
HCX	1.73	15.90
3B15	0.92	6.11
68020	1.50	12.79
32016	1.32	10.67
80386	1.09	7.60
3230	0.81	5.47
RT	2.04	11.68
CLIPPER	1.60	13.89
SPARC	0.08	0.00

Table 7: Number of Saves and Restores of Allocable Registers

The number of saves and restores are affected by the number of total allocable registers and the number of scratch registers as shown in Table 8. They are also affected by how registers can be used. There are twelve registers on the 3230 and RT and eight registers on the 68020, 32016, 80386, and Clipper that can only be used to hold floating-point values. The number of allocable registers on the SPARC is represented by the number of registers accessible from a single function (window). Of the forty-five scratch registers, thirty-two are floating-point. With the eight windows on the SPARC there are actually 151 allocable registers.

Machine	Scratch	Nonscratch	Total
VAX-11	6	6	12
HCX	2	11	13
3B15	6	6	12
68020	6	15	21
32016	3	13	16
80386	3	11	14
3230	23	0	23
RT	16	8	24
CLIPPER	10	13	23
SPARC	45	14	59

Table 8: Allocable Machine Registers

3.4.1.7. Register Usage

The number of registers used per call varied depending mostly on the number of registers that were available to the register allocator. The RISC machines, the RT, Clipper, and SPARC, were unique in having more registers allocated to arguments than to local variables. This was due to the RT, Clipper, and SPARC calling sequences which allow up to four, two, and six arguments respectively to be passed in registers. The RT was unusual in the higher number of registers used as temporaries per call. This was partially caused by the unusual RT calling sequence requiring the use of three registers as temporaries. Since the RISC machines only reference memory via load and store instructions, all calculations had to be performed in registers. Referencing a scratch register used as an argument also was counted as a temporary use. This resulted in more registers used as temporaries. Register usage per call is shown in Table 9.

Machine	Locals	Arguments	Temporaries	Total
VAX-11	1.22	0.49	1.39	3.10
HCX	1.25	0.48	1.49	3.22
3B15	1.06	0.47	1.60	3.14
68020	1.22	0.43	1.93	3.57
32016	1.10	0.49	1.62	3.21
80386	0.78	0.24	1.91	2.93
3230	0.88	0.33	1.97	3.18
RT	1.16	1.35	5.65	8.16
CLIPPER	1.25	1.26	2.43	4.94
SPARC	1.29	1.75	4.98	8.02

Table 9: Register Usage Per Call

One may conclude from the above table that only three or four allocable registers are needed for a machine such as the VAX-11. Though the average number of registers used per call was 3.10, the standard deviation was 2.31. Figure 17 shows there are several instances when additional registers are used.

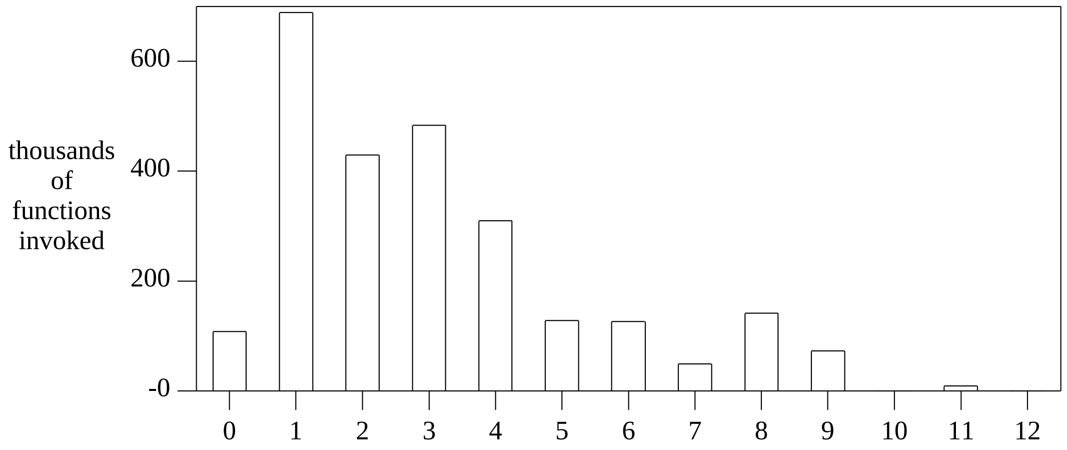


Figure 17: Distribution of Total VAX-11 Registers Used Per Call

3.4.1.8. Condition Code Usage

Disregarding the instructions whose purpose is to set the condition codes and have no other effect (e.g. VAX-11: `bit`, `cmp`, `tst`), condition codes were not used very often. The results of using condition codes that are set as side-effects is shown in Table 10. Setting the condition codes as a side-effect for the SPARC is explicitly specified as an option for arithmetic and data movement instructions. There are no instructions that set the condition codes as a side-effect for the 32016. Also data movement instructions, which set condition codes as side-effects on other machines, do not set the condition codes on the 80386, RT, or the Clipper. Since the 3230 compares a value to zero by loading it into a register, there was a higher use of condition code side-effects on the 3230.

Machine	Per Cent CC Side Effect Used
VAX-11	2.79
HCX	2.29
3B15	2.61
68020	3.04
32016	0.00
80386	0.58
3230	6.76
RT	0.66
CLIPPER	0.26
SPARC	0.00

Table 10: Usage of Condition Codes Set as Side Effects

3.4.1.9. Branch Instructions

Conditional branches were taken consistently on all machines about two thirds of the time. The average number of instructions between branches (and calls) varied depending on the complexity of the instructions on the different architectures as shown in Table 11.

Machine	Instructions Executed Between Branches	Instructions Executed Between Calls
VAX-11	3.81	30.67
HCX	4.29	33.62
3B15	5.12	40.06
68020	5.23	42.30
32016	5.17	40.16
80386	5.99	45.90
3230	7.68	57.05
RT	10.14	67.38
CLIPPER	7.23	55.10
SPARC	7.88	47.28

Table 11: Average Number of Instructions Executed between Branches

3.4.1.10. Data Type Distribution

LONGs were used extensively by all the architectures. Of instructions associated with a data type, the percentage of instructions manipulating LONGs was very high. This occurred since the default integer type for C on each of the architectures in the study was a LONG (32 bits). The percentage of long operand references from the total number of operand references was not quite as high as instructions manipulating LONGs. Some of the more complicated addressing modes had a greater proportion of other data types. The only addressing mode not referenced as a LONG most of the time was the auto-increment addressing mode on the VAX-11 and 68020. For this addressing mode, the byte data type was used more frequently due to the loops involving character processing in some of the test set programs.

3.4.1.11. Instruction Field Distribution

Many of the instruction fields used only a small subset of the available addressing modes. To allow any addressing mode to be used in each operand of each type of instruction simplifies the task of the compiler since there are few special cases. If the cost of the machine can be reduced or the speed of the machine increased by not allowing infrequently used addressing modes for specific operands of specific instructions, then perhaps the reduced cost and/or increased speed of the architecture might outweigh the increased cost of the compiler.

3.4.1.12. Summary of Dynamic Measurement Analysis

There are several observations that can be made from the analysis of dynamic measurements. The machines with the least number of instructions had both the ability to reference three unique operands and to reference memory in each field of each instruction. These machines, despite having more bytes per instruction, had fewer total number of bytes of instructions that were executed. This suggests if reducing code size is a high priority, then three-address instructions and the ability to reference memory in each field are important architectural features.

It was found that both simpler instructions and addressing modes were used most frequently. Data movement instructions, typically simple instructions, were the most frequently used class of instructions for most machines. The register addressing mode dominated all of the machines in the study. These results support the RISC contention that complex instructions and addressing modes are rarely used and may be a liability in an instruction set [Pat85].

Both operations and operands dealt with mostly LONGs. Several of the machines in the study, including all of the RISCs, do not support operations directly on bytes or halfwords. For these machines only values in memory were allowed to contain bytes or halfwords. Though byte or halfword arithmetic executes more quickly than longword arithmetic, performing the operations in longwords was beneficial since more effective encoding could occur.

The number of memory references were affected by many factors. One factor is the number of allocable registers. For instance, the fixed-point portion of the VAX-11 and HCX-9 architectures are very similar. The HCX-9 has thirteen allocable registers and the VAX-11 has only twelve. This difference of one register resulted in slightly more variables being allocated to registers for the HCX-9 and more saves and restores. The HCX-9 also had 9% fewer bytes transferred to and from memory than the VAX-11. The method for saving and restoring registers also can affect the number of memory references. The only machine with a caller-save method was the 3230, which had the fewest variables allocated to registers and the most memory references. Another influential factor was the method for handling function linkage. For instance, almost 25% of the VAX-11 memory references resulted from function linkage. The Clipper, which used much more primitive call and return instructions, required only one fourth the number of memory references for function linkage as the VAX-11. This suggests that using primitive call and return instructions may be more effective than using their complex counterparts. Constructing global addresses on the SPARC was accomplished by referencing the high and low portions of the address in two separate instructions. About 12% of the memory references for the RT were a result of their method of loading a global address from a constant pool. This did not reduce the number of instructions since a separate load instruction was required to get the address. Thus, it seems that the method of constructing a global address used on the SPARC for machines that cannot reference a global address directly is superior to using a constant pool since no memory reference is required.

There are several interesting observations that can be made concerning register usage. Machines that passed arguments through registers resulted in substantially more arguments being allocated to registers. This phenomenon is examined in an argument-passing experiment described later in this dissertation. The average number of registers per call referenced for most machines was quite small, yet the deviation was high. Since there are many instances when several registers can be used, this suggests that the number of registers available should be greater than the average used per call. A final observation is that machines with the ability to reference memory in each field of each instruction resulted in fewer registers used as temporaries. Fewer references to temporaries occurred since calculations by complex arithmetic instructions could reference memory directly. Thus, the most effective percentage of registers that should be designated as scratch on a CISC machine may be less than on a RISC machine.

Types of branch instructions varied little with each architecture. This resulted in a similar number of branch instructions executed for each architecture. Machines with more complex instructions had fewer instructions of other classes and thus had fewer instructions between branches. This observation implies that CISC machines would have fewer instructions to place behind delayed branches. Machines with condition codes that are set as side-effects of instructions were rarely used by conditional branches. A machine architect should consider the increased cycle times of instructions, the space and cost in firmware or hardware, and the added complexity before designing instructions that set condition codes as side-effects. Since conditional branches were consistently taken about two thirds of the time, instructions at the target of a conditional branch would be useful more often to fill the delay slot of a conditional branch than instructions following the branch. This supports the SPARC delayed branch with the annul option, which only executes the instruction following the branch if the branch is taken.

There were also some machine-independent observations. One was that 56% of the functions invoked were leaves. Functions that do not call other functions are known as leaf functions. On average leaf functions had fewer variables and arguments than non-leaf functions. These observations can influence the design of the method for saving and restoring registers, passing arguments through registers, and the complexity of call and return instructions. The impact of a high frequency of leaf functions on these calling sequence conventions are examined later in this dissertation.

3.4.2. Static Versus Dynamic Measurements

Dynamic measurements are generally considered to give more accurate data than static measurements for measuring the influence of architectural features [Kat83, SwS82]. Comparisons between static and dynamic measurements collected from the execution of the test set showed that for each of the machines there are consistent differences between the results of the two types of measurements.

3.4.2.1. Instruction Type Distribution

Consistently on all the different machines, the dynamic measurements had a greater percentage of compare and conditional jump instructions. There was, however, a smaller percentage of call and unconditional jump instructions than the static measurements. This is shown in Table 12.

Use of conditional and looping constructs increases the dynamic percentage of compare and conditional jump instructions. The compare and conditional jump instructions generated from an `if` statement will be executed twice the number of times of the average of the instructions generated from the `then` block and `else` block. Also, compare and conditional jump instructions are generated inside of loops from the compilation of the test condition in looping constructs.

Use of conditional and looping constructs decreases the dynamic percentage of unconditional jump instructions. An unconditional jump instruction is generated at the end of the `then` block of an `if-then-else` construct to jump across the `else` block. Unconditional jumps are also used to imple-

Machine	Compares Pct Static	Compares Pct Dynamic	Compares Diff	Conditional Jumps Pct Static	Conditional Jumps Pct Dynamic	Conditional Jumps Diff
VAX-11	11.88	16.37	4.49	13.53	18.05	4.52
HCX	11.42	14.98	3.56	12.88	16.49	3.61
3B15	11.57	13.81	2.24	12.78	15.06	2.28
68020	9.28	12.03	2.75	10.93	13.86	2.93
32016	11.18	14.70	3.52	11.19	14.70	3.51
80386	9.96	12.64	2.68	10.04	12.72	2.68
3230	4.73	5.70	0.97	7.97	10.66	2.69
RT	6.81	7.21	0.40	7.38	7.90	0.52
CLIPPER	8.98	9.89	0.91	9.96	10.59	0.63
SPARC	4.89	5.25	0.36	8.27	9.60	1.33
Machine	**Calls Pct Static**	**Calls Pct Dynamic**	**Calls Diff**	**Unconditional Jumps Pct Static**	**Unconditional Jumps Pct Dynamic**	**Unconditional Jumps Diff**
VAX-11	10.15	3.16	-6.99	5.65	2.82	-2.83
HCX	9.81	2.89	-6.92	5.24	2.16	-3.08
3B15	9.43	2.44	-6.99	5.16	1.95	-3.21
68020	8.14	2.31	-5.83	5.97	2.54	-3.43
32016	8.35	2.43	-5.92	6.03	2.54	-3.49
80386	7.35	2.13	-5.22	5.50	2.04	-3.46
3230	5.96	1.72	-4.24	3.24	1.43	-1.81
RT	5.79	1.46	-4.33	3.95	1.17	-2.78
CLIPPER	7.42	1.78	-5.64	5.45	1.69	-3.76
SPARC	6.49	2.07	-4.42	4.59	1.71	-2.88

Table 12: Static and Dynamic Measurements of Compare, Call, and Jump Instructions

ment `break` statements inside `switch` constructs. Since the unconditional jumps generated in these situations are not always executed each time an `if-then-else` or `switch` statement is entered, the percentage of dynamic unconditional jump instructions is decreased. Unconditional jumps are also generated from the compilation of looping constructs. If the termination condition is not known for the first iteration of a `for` loop construct, then an unconditional jump is generated to jump to the bottom of the loop where the termination condition is tested. Unconditional jumps are also used to implement `break` statements to exit loops. Thus, the percentage of unconditional jump instructions is decreased since the unconditional jumps in these two situations are at most only executed once each time the loop is entered.

In many typical programs there are functions that are only called in special circumstances, such as error handling routines. In other cases some functions are invoked infrequently. For instance, functions to read and write data may only be invoked early and late respectively in the execution of the program. Thus, these functions may be invoked outside the loops which are responsible for most of the dynamic instructions. Leaf functions are often invoked from more than one place. The static percentage of functions entered that were leaf functions was only 17%. The dynamic percentage of functions that were leaves was 56%. Since for typical executions many functions are infrequently or never called and leaf functions (those with no call instructions) are invoked more frequently, the percentage of dynamic call instructions is reduced.

Differences between the percentages of static and dynamic return instructions, shown in Table 13, varied depending upon the code generation strategy used by the compiler for the machine. The first code generation strategy, used by the VAX-11, HCX, and 3230 compilers, generated one or more static return instructions per function. Since at most one return can be executed each time a function is entered, the dynamic percentage of instructions that were returns was less than the static percentage. The compilers for the other machines generated at most one static return instruction per function since other instructions were required to restore the state of the calling function. In order to use less static space for the program, an unconditional jump to the return sequence was often required. Using the second code generation strategy resulted in an increase from the static to dynamic percentage of return instructions.

Machine	Pct Static	Pct Dynamic	Diff
VAX-11	4.08	2.82	-1.26
HCX	4.00	2.58	-1.42
3B15	3.86	2.18	-1.68
68020	1.61	2.06	0.45
32016	1.65	2.19	0.54
80386	1.44	1.90	0.46
3230	2.39	1.43	-0.96
RT	1.13	1.20	0.07
CLIPPER	1.46	1.59	0.13
SPARC	1.18	1.38	0.20

Table 13: Static and Dynamic Measurements of Returns

The integer add and subtract operations are frequently used arithmetic instructions. Add instructions are commonly generated from compilation of the iteration portion of a `for` loop construct. Because instructions inside of loops are executed more often than instructions outside of loops, the percentage of arithmetic instructions is increased. The static and dynamic percentages of add and subtract instructions are shown in Table 14. Note that the percentages of add instructions on the 68020 and 80386 were adjusted to exclude instructions used to adjust the stack pointer after a call. Since the dynamic percentage of call instructions is smaller than the static percentage, an architecture that uses an add instruction to adjust the stack pointer after a call will result in a smaller increase in the dynamic percentage over the static percentage of add instructions. The static and dynamic percentages of subtract operations were less predictable. These percentages depended upon many factors that include the direction that the run-time stack grows when allocating space for local variables on the stack, the instruction used to allocate space on the stack, and the availability of other instructions that may accomplish a subtract operation through use of a displacement addressing mode.

Machine	Integer Adds			Integer Subs		
	Pct Static	Pct Dynamic	Diff	Pct Static	Pct Dynamic	Diff
VAX-11	6.65	13.72	7.07	2.96	3.19	0.23
HCX	7.15	14.49	7.34	4.07	5.32	1.25
3B15	8.90	19.55	10.65	2.30	2.21	-0.09
68020	3.13	7.49	4.36	1.57	2.10	0.53
32016	2.30	4.19	1.89	0.47	0.75	0.28
80386	2.53	6.10	3.57	1.72	2.28	0.56
3230	3.30	7.59	4.29	0.99	1.68	0.69
RT	8.25	15.53	7.28	0.65	2.28	1.63
CLIPPER	4.06	8.88	4.82	2.15	1.95	-0.20
SPARC	9.70	14.28	4.58	1.67	3.36	1.69

Table 14: Static and Dynamic Measurements of Integer Adds and Subtracts

The static and dynamic measurements for integer multiply and divide instructions were also examined. These measurements are shown in Table 15. The compiler for the RT either converts a multiply by an integer constant into a series of shifts, adds, and subtracts or performs 16 multiply steps. The compiler for the SPARC also converts multiplies by an integer constant. Multiplies by a register are implemented by invoking a library function. Divides on both the RT and SPARC are implemented by invoking a library function. Therefore, measurements of the multiply and divide instruction for the RT and SPARC were not included.

Machine	Integer Multiplies			Integer Divides		
	Pct Static	Pct Dynamic	Diff	Pct Static	Pct Dynamic	Diff
VAX-11	0.77	2.61	1.84	0.52	0.34	-0.18
HCX	0.76	2.36	1.60	0.51	0.30	-0.21
3B15	0.57	1.91	1.34	0.35	0.22	-0.13
68020	0.50	1.84	1.34	0.43	0.24	-0.19
32016	0.52	1.96	1.44	0.43	0.25	-0.18
80386	0.47	1.67	1.20	0.39	0.24	-0.15
3230	0.37	1.35	0.98	0.31	0.18	-0.13
RT	N/A	N/A	N/A	N/A	N/A	N/A
CLIPPER	0.46	1.40	0.94	0.38	0.18	-0.20
SPARC	N/A	N/A	N/A	N/A	N/A	N/A

Table 15: Static and Dynamic Measurements of Integer Subtracts and Multiplies

Table 16 shows the static and dynamic percentages of logical instructions to accomplish bitwise and, bitwise or, and shift instructions. A left shift instruction is often used to implement array indexing. Array elements are typically processed in a loop. Thus, the dynamic percentage of shift instructions was larger than the static percentage of shift instructions. The percentages of bitwise and instructions were less predictable since the bitwise and operation was used for a variety of purposes on different machines. For instance, on the RT a bitwise and operation was required to accomplish a floating-point comparison. Some machines used the bitwise and operation to extract bit fields and conversion to smaller integral types. The bitwise or instruction for the SPARC was used in constructing global addresses and constants requiring more than twelve bits.

Machine	Bitwise Ands			Bitwise Ors			Shifts		
	Pct Static	Pct Dyn	Diff	Pct Static	Pct Dyn	Diff	Pct Static	Pct Dyn	Diff
VAX-11	1.78	2.26	0.52	0.45	0.11	-0.34	1.28	4.55	3.27
HCX	2.02	2.44	0.42	0.44	0.10	-0.34	1.60	5.69	4.09
3B15	1.26	1.33	0.07	0.43	0.09	-0.34	2.90	10.55	7.65
68020	1.94	2.12	0.18	0.37	0.08	-0.29	0.44	1.48	1.04
32016	1.85	2.05	0.20	0.38	0.09	-0.29	0.25	1.53	1.28
80386	0.95	1.04	0.09	0.34	0.08	-0.26	0.31	1.33	1.02
3230	0.79	0.86	0.07	0.27	0.06	-0.21	2.05	7.61	5.56
RT	1.13	0.97	-0.16	0.24	0.04	-0.20	2.09	8.10	6.01
CLIPPER	1.52	1.34	-0.18	0.34	0.06	-0.28	2.07	7.61	5.54
SPARC	1.31	1.21	-0.10	4.47	4.68	0.21	2.29	10.04	7.75

Table 16: Static and Dynamic Measurements of Ands, Ors, and Shifts

The static and dynamic measurements for floating-point operations were also examined. Though floating-point adds, subtracts, multiplies, and divides were not used frequently, the direction of increase or decrease between the static and dynamic percentages was consistent on all machines. These measurements are shown in Table 17.

Machine	Float Adds			Float Subtracts		
	Pct Static	Pct Dynamic	Diff	Pct Static	Pct Dynamic	Diff
VAX-11	0.15	0.44	0.29	0.11	0.06	-0.05
HCX	0.15	0.39	0.24	0.11	0.05	-0.06
3B15	0.14	0.33	0.19	0.10	0.05	-0.05
68020	0.12	0.34	0.22	0.09	0.04	-0.05
32016	0.12	0.34	0.22	0.09	0.04	-0.05
80386	0.12	0.29	0.17	0.08	0.04	-0.04
3230	0.09	0.18	0.09	0.06	0.01	-0.05
RT	N/A	N/A	N/A	N/A	N/A	N/A
CLIPPER	0.11	0.24	0.13	0.08	0.03	-0.05
SPARC	0.09	0.21	0.12	0.06	0.03	-0.03
Machine	Float Multiples			Float Divides		
	Pct Static	Pct Dynamic	Diff	Pct Static	Pct Dynamic	Diff
VAX-11	0.18	0.28	0.10	0.09	0.12	0.03
HCX	0.17	0.25	0.08	0.08	0.11	0.03
3B15	0.16	0.21	0.05	0.07	0.09	0.02
68020	0.14	0.20	0.06	0.07	0.09	0.02
32016	0.15	0.22	0.07	0.07	0.09	0.02
80386	0.13	0.18	0.05	0.06	0.08	0.02
3230	0.11	0.12	0.01	0.05	0.06	0.01
RT	N/A	N/A	N/A	N/A	N/A	N/A
CLIPPER	0.13	0.15	0.02	0.06	0.07	0.01
SPARC	0.11	0.13	0.02	0.05	0.06	0.01

Table 17: Static and Dynamic Measurements of Floating-Point Instructions

3.4.2.2. Addressing Mode Distribution

There were consistent differences between dynamic and static measurements of addressing modes across all the machines examined. Dynamic measurements had a greater percentage of register mode usage and a smaller percentage of immediate mode usage as shown in Table 18.

Machine	Register Addr Mode			Immediate Addr Mode		
	Pct Static	Pct Dynamic	Diff	Pct Static	Pct Dynamic	Diff
VAX-11	29.64	40.57	10.93	23.99	19.29	-4.70
HCX	34.53	47.53	13.00	23.39	19.11	-4.28
3B15	32.27	49.87	17.60	24.17	18.16	-6.01
68020	41.42	50.81	9.39	17.22	14.14	-3.08
32016	41.97	44.95	2.98	19.52	15.72	-3.80
80386	42.47	50.08	7.61	14.38	10.25	-4.13
3230	49.47	60.25	10.78	13.79	12.28	-1.51
RT	58.67	69.54	10.87	19.49	16.38	-3.11
CLIPPER	57.45	65.55	8.10	13.53	11.43	-2.10
SPARC	54.96	64.62	9.66	24.54	18.62	-5.92

Table 18: Static and Dynamic Measurements of Register and Immediate Modes

Typically, instructions in loops are executed more frequently than instructions which are not in loops. Immediate values are often used to initialize variables outside of loops. This reduces the percentage of immediate mode usage in dynamic measurements. *Vpo* used a cost function to attempt to place the most frequently used variables in registers. The cost function weighted memory references in loops more heavily. Since variables with the most estimated memory references were assigned to registers, there were many register references inside of loops which increased the percentage of register mode usage in dynamic measurements.

Other addressing modes used on most machines include register deferred, displacement, and direct modes. The static and dynamic measurements for these modes are shown in Table 19. Note that there is no direct addressing mode on the RT or SPARC. Addresses of globals on the RT were stored in a constant pool and were referenced using displacement and register deferred modes. Addresses of globals on the SPARC were constructed by two instructions. The dynamic percentage use of the displacement mode on the 3230 was smaller than the static percentage since the displacement mode was used in saving and restoring registers around call instructions. The displacement mode was also used for pushing arguments on the run-time stack for the RT, CLIPPER, and SPARC. Because the dynamic percentage of call and push instructions was smaller than the static percentage, the dynamic use of displacement mode was also smaller. The displacement mode also varied for the SPARC since displacements could represent at most twelve bits.

Machine	Register Deferred			Displacement			Direct		
	Pct Static	Pct Dyn	Diff	Pct Static	Pct Dyn	Diff	Pct Static	Pct Dyn	Diff
VAX-11	3.43	4.08	0.65	6.87	8.41	1.54	15.52	7.65	-7.87
HCX	3.31	4.35	1.04	5.36	6.21	0.85	14.66	6.69	-7.97
3B15	4.13	6.80	2.67	7.97	9.20	1.23	14.56	5.68	-8.88
68020	2.49	3.27	0.78	6.85	7.24	0.39	14.21	7.14	-7.07
32016	2.58	4.28	1.70	8.55	10.57	2.02	20.26	8.71	-11.55
80386	2.59	3.35	0.76	10.37	13.25	2.88	11.66	5.51	-6.15
3230	3.41	4.47	1.06	19.27	11.42	-7.85	7.95	3.52	-4.43
RT	6.68	6.09	-0.59	14.98	7.80	-7.18	N/A	N/A	N/A
CLIPPER	3.73	5.74	2.01	6.68	5.90	-0.78	10.14	3.68	-6.46
SPARC	2.26	3.11	0.85	7.72	5.07	-2.65	N/A	N/A	N/A

Table 19: Static and Dynamic Measurements of Other Addressing Modes

3.4.2.3. Bytes per Instruction

On all the machines, except the SPARC, the dynamic measurements consistently resulted in a smaller number of bytes per instruction than the static measurements. The static and dynamic number of bytes per instruction were the same since the architects of the SPARC designed every instruction to be the same size (four bytes). This is shown in Table 20. As noted earlier, the dynamic measurements showed a smaller percentage of instruction operands referencing memory and a greater percentage of instruction operands referencing registers. Memory references require more bits in an operand field than register references. This decreases the number of bytes per instruction in dynamic measurements. Note that the number of bytes per instruction for the 3B15 was larger than the other machines since code was generated for the machine-independent instruction set. Some of these instructions expanded into several machine-dependent instructions. The difference between the static and dynamic bytes per instruction is of course dependent on the scheme used for encoding instructions on each machine.

Machine	Static	Dynamic	Diff
VAX-11	4.41	4.14	-0.27
HCX	4.40	3.98	-0.42
3B15	5.66	4.15	-1.51
68020	4.00	3.52	-0.48
32016	3.98	3.52	-0.46
80386	3.52	3.15	-0.37
3230	4.22	3.61	-0.62
RT	3.01	2.59	-0.42
CLIPPER	4.14	3.61	-0.53
SPARC	4.00	4.00	0.00

Table 20: Bytes per Instruction

3.4.2.4. Memory References

There were a few differences in the measurements of memory references that were consistent for the machines examined. On all the architectures in this study there was a greater percentage of dynamic byte memory references than static byte memory references due to the loops involving character processing for several programs in the test set. The dynamic measurements had a smaller percentage of operands referencing memory than the static measurements due to variables used heavily inside of loops being allocated to registers.

3.4.2.5. Register Usage

Dynamic measurements consistently showed a greater percentage use of scratch registers (as opposed to non-scratch registers) and a smaller number of saves and restores per call on all the architectures in the study as shown in Table 21. It is interesting that the 3230, the only machine that used a caller-save method for saving and restoring general-purpose registers, was also the only machine that had more dynamic than static saves and restores. Since the 3230 used a caller-save method, all of its registers are scratch.

Machine	Saves/Restores per Call			Per Cent Use of Scratch Registers		
	Static	Dynamic	Diff	Static	Dynamic	Diff
VAX-11	1.60	1.19	-0.41	47.12	61.49	14.37
HCX	2.16	1.73	-0.43	34.61	46.13	11.52
3B15	1.38	0.92	-0.46	39.46	56.40	16.94
68020	1.97	1.50	-0.47	47.68	58.12	10.44
32016	1.63	1.32	-0.31	44.25	54.34	10.09
80386	1.36	1.11	-0.25	53.45	62.74	9.29
3230	4.55	4.82	0.27	100.00	100.00	0.00
RT	4.19	4.04	-0.15	55.22	57.75	2.53
CLIPPER	2.01	1.60	-0.41	57.27	67.70	10.43
SPARC	N/A	N/A	N/A	52.47	55.64	3.17

Table 21: Static and Dynamic Measurements of Register Information

Leaf functions are often used as utility or library functions and are invoked from more than one place in the program. They are also often invoked from a loop in non-leaf functions. Therefore, leaf functions are usually invoked more frequently than non-leaf functions. Only 17% of the functions in the test set were leaf functions, yet 56% of the invoked functions were leaves. *Vpo* allowed scratch registers in leaf functions to be allocated to local variables and arguments. Therefore, dynamic measurements for machines with callee-save methods showed a greater percentage use of scratch registers and a smaller number of saves and restores per call. Note that there are no saves and restores in a leaf function using a caller-save method.

3.4.2.6. Correlations and Regression Models

Typically static measurements are easier to obtain than dynamic measurements. By examining the differences between static and dynamic measurements, one could estimate dynamic measurements from static measurements on other architectures [DRW89]. If it is determined that there is a high linear correlation between a static and dynamic measurement, then a linear equation can be derived that can effectively use a static measurement to predict the corresponding dynamic measurement. A statistical package, SPSS [Nor86], was used to determine the correlation between static and dynamic measurements and to produce an estimator of dynamic measurements using regression analysis. The static-dynamic percentage pairs for each machine were used as input to SPSS to compute the Pearson correlation coefficient, an indicator of linear relationship. This coefficient can have a possible range from -1 to 1. When the value is near either extreme of the range, a linear relationship is suggested.

For instance, the static-dynamic pairs for bitwise and instructions from Table 16 resulted in a Pearson correlation coefficient of 0.9517. The regression analysis method of least squares was used to determine a line that minimizes the sum of squared distances from the observed data points representing static-dynamic pairs from that line. This regression line is represented by the equation

$$dynamic = b_0 static + b_1$$

where $b_0 = 0.33$ and $b_1 = -0.43$ are the linear coefficients from least-squares fitting. This line is shown along with the data points representing static-dynamic pairs of bitwise and instruction percentages in Figure 18.

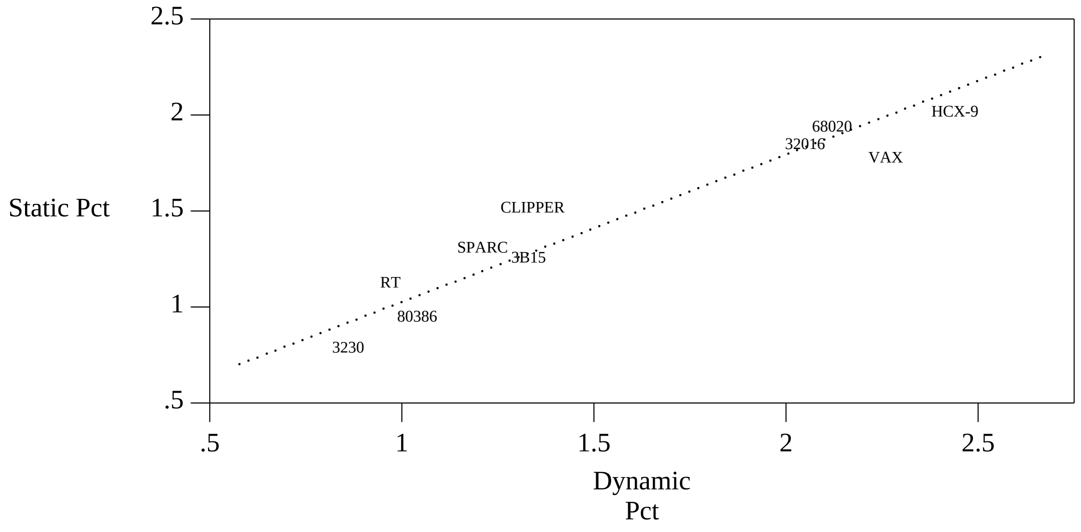


Figure 18: Percentage of Bitwise And Instructions

In the presentation of correlation coefficients in Table 22, asterisks are used to designate those for which the data support rejection of a hypothesis of no correlation at a very strong significance level. Significance level is the probability that a sampling of values of uncorrelated random variables would yield results at least as suggestive of a linear relationship as those obtained. In Table 22 one asterisk indicates that the data give sufficient evidence to reject a one-sided hypothesis at a significance level of .01, while two asterisks indicate rejection at a .001 significance level. Regression analysis achieves an estimator of each dynamic measurement consisting of a constant multiple of the static measurement plus a constant. Table 22 also shows the estimate of the constant coefficient (b_0) by which a given static measurement is to be multiplied as well as that for the additive constant (b_1). The measurements for unconditional jumps and returns were broken into two extra categories. The first category indicates that several return instructions can appear within a function. The second category only allows one return in a function. The second category was used for machines that required additional instructions before the return to restore the state of the calling function. At each point a return was needed, an unconditional jump was generated to the bottom of the function. This resulted in a smaller number of static instruction bytes. The RT and the SPARC were not used for multiplies and divides since these machines did not have these instructions. The RT was not used for the register deferred, displacement, and direct addressing modes since direct addresses were implemented as register deferred and displacement. The SPARC was also not used for the displacement and direct addressing modes. A direct address on the SPARC is constructed by two instructions. The displacement addressing mode only allowed displacements that could be represented in twelve bits. The RT was not included in the floating-point instructions since floating-point operations were implemented using memory-mapped I/O. The SPARC was not included for saves and restores since this machine uses register windows. The SPARC was also not included in the sample for bytes per instruction since all instructions on the SPARC are four bytes in length. Note also that the measurements from the 3230 were omitted for displacement addressing mode, saves/restores per call, and use of scratch registers since a caller-save calling sequence was used.

Measurement	Num Cases	Correlation Coefficient	Coefficient b_0	Coefficient b_1
Instructions				
calls	10	.9323**	0.31	-0.23
compares	10	.9814**	1.46	-1.95
conditional jumps	10	.9597**	1.43	-2.09
unconditional jumps - 1	4	.8775	0.47	-0.17
unconditional jumps - 2	6	.9233*	0.60	-1.22
returns - 1	4	.9445	0.72	-0.32
returns - 2	6	.9530*	1.74	-0.73
integer adds	10	.9401**	1.65	1.95
integer subtracts	9	.8404*	0.95	0.76
integer multiplies	8	.9564**	2.91	0.28
integer divides	8	.9229*	0.69	-0.04
shifts	10	.9876**	3.74	0.13
bitwise ands	10	.9517**	1.30	-0.33
bitwise ors	9	.9435**	0.28	-0.02
float add	9	.9435**	3.51	-0.12
float subtract	9	.9044**	0.70	-0.02
float multiply	9	.9730**	2.11	-0.11
float divide	9	.9899**	1.55	-0.02
Address Modes				
register addrmode	10	.9333**	0.88	15.63
immediate addrmode	10	.9710**	0.72	1.48
register deferred addrmode	9	.8973**	1.69	-0.85
displacement addrmode	7	.9830**	1.45	-1.85
direct addrmode	8	.9402*	0.46	-0.22
Miscellaneous				
saves/restores per call	8	.9962*	1.08	-0.52
use of scratch registers	9	.7870*	0.63	27.82
bytes per instruction	8	.9771**	0.98	-0.38

Table 22: Correlations and Linear Estimators

This line, together with other statistics, can be used to derive a confidence interval estimate of the population mean dynamic figure for any given static figure within the static range of our sample. The formula [BLG83] for such an estimate is:

$$b_0 static_{given} + b_1 \pm t_{1-\alpha/2;n-2}\, S_e \sqrt{\frac{1}{n} + \frac{(static_{given} - static_{mean})^2}{(n-1)S_{static}{}^2}}$$

where

$b_0,\ b_1$	are the linear coefficients for the least-squares fit;
$static_{given}$	is the given static level at which prediction is desired;
$1-\alpha$	is the degree of confidence associated with the interval estimate;
$t_{\gamma\,;\,df}$	is the abscissa value associated with the standard Student-t distribution with df degrees of freedom below which lies 100γ percent of the area under the distribution curve;
S_e	is the standard error of the estimate;
n	is the sample size;
$static_{mean}$	is the mean of the static sample figures; and
S_{static}	is the static sample standard deviation.

So, for instance, to estimate the true mean dynamic `cmp` statement percentage given a static `cmp` percentage of 9.17, a static sample mean of 9.07, a static sample standard deviation of 2.71, and a standard error of the estimate of 0.82, we might use the results shown in Table 22 and the formula above to get a 95 percent confidence interval of

$$1.46\,(9.17) - 1.95 \pm (2.31)(0.82)\sqrt{\frac{1}{10} + \frac{(9.17 - 9.07)^2}{9(2.71)^2}}$$

or about

$$11.44 \pm 0.60.$$

Note that this is an estimate of the population mean rather than an estimate of an individual outcome. An individual prediction interval would be considerably wider and is found by a formula similar to that given for the mean above.

$$b_0 static_{given} + b_1 \pm t_{1-\alpha/2;n-2}\, S_e \sqrt{1 + \frac{1}{n} + \frac{(static_{given} - static_{mean})^2}{(n-1)S_{static}{}^2}}$$

Calculation of an individual 95% confidence interval of the `cmp` instruction results in the interval

$$11.44 \pm 1.99.$$

Note, too, that such estimates are appropriate only within the sample static ranges for each measure.

To test if the coefficients shown in Table 22 could accurately predict dynamic measurements from their static counterparts, a 95% individual confidence interval for each category was calculated. A 95%

confidence interval for a specific category indicates that there is a 95% chance that given a static value, the dynamic value would be in that specified range. The input data to the test programs for the 68020 was changed, the test programs executed, and new measurements collected. Only one of the twenty-four categories of measurements, bitwise ands, resulted in a dynamic value that fell outside of its confidence interval. Thus, given a large enough test set, these linear relationships are not totally dependent on the test data which determine the execution paths taken. The low and high ranges of the confidence interval and the actual dynamic value from the new measurements on the 68020 are shown in Table 23.

Measurement	Low	High	Actual
Instructions			
calls	2.18	3.20	2.34
compares	9.45	13.43	12.39
conditional jumps	11.06	15.76	14.87
unconditional jumps - 2	1.63	3.11	2.55
returns - 2	1.57	2.37	2.12
integer adds	2.39	11.45	7.40
integer subtracts	0.43	3.95	1.91
integer multiplies	1.27	2.09	1.99
integer divides	0.16	0.32	0.30
shifts	-0.05	3.37	1.11
bitwise ands	2.00	3.04	1.91
bitwise ors	0.06	0.10	0.10
float add	0.23	0.37	0.33
float subtract	0.02	0.06	0.04
float multiply	0.17	0.21	0.21
float divide	0.04	0.10	0.09
Address Modes			
register addrmode	43.01	62.23	50.10
immediate addrmode	11.57	15.73	13.95
register deferred addrmode	2.25	5.17	3.01
displacement addrmode	5.91	9.73	6.77
direct addrmode	4.51	8.05	7.28
Miscellaneous			
saves/restores per call	-0.83	4.05	1.33
use of scratch registers	48.00	67.72	60.96
bytes per instruction	3.24	3.82	3.52

Table 23: Confidence Intervals and Results for the 68020

3.4.3. Comparing the Architectures

The actual time required to perform each type of instruction on a given machine depends upon the actual hardware implementation. Some vendors do not supply the execution times for each instruction due to privacy or difficulty in obtaining exact figures. Using execution times to compare architectures may result in misleading conclusions. In the CFA studies [FuB77] the performance of the architectures were evaluated by only three simple measures.

1. S : Number of static bytes used to represent a program.
2. M : Number of bytes transferred to and from memory.
3. R : Number of bytes transferred between registers.

Perhaps part of the reason for using such simple measures was the difficulty of collecting more detailed information. This architectural study does not suffer from that limitation.

Another use of the obtained measurements is to compare the effectiveness of the architecture of each machine by attempting to determine the total architectural cost of the execution of the test set. To only compare architectures and not hardware implementations, each machine instruction was associated with a generic instruction. For instance, the ‘‘`add2`’’ instruction on the VAX-11 and the ‘‘`a`’’ instruction on the RT were both associated with the generic `ADD_INST` instruction. Each generic instruction is assigned a cost. Similarly every addressing mode on each machine was associated with a generic addressing mode with a specified cost. These costs were derived by examining the relative times of different types of operations and addressing modes published from different vendors.

3.4.3.1. Instruction Costs

The instruction cost results are shown in Figure 19 and 20. The dotted area represents the instruction costs when delayed branches are not taken into account. This area is calculated by assuming that a portion of the execution of each instruction following a delayed branch is overlapped with the execution of the delayed branch. This portion is the execution cost of a move instruction. The dashed area represents the instruction costs when delayed loads are not taken into account. This area is also calculated by assuming that the execution cost of a move instruction is overlapped with the execution of the delayed load. The high total of instruction costs for the 3230 was partially due to saving and restoring registers by its caller-save method, which accounted for over 10.5% of the instruction costs. These figures show the effectiveness of pipelining loads and branches for the three RISC machines. Figure 20 illustrates that the machines with more complicated instructions had a greater cost per instruction.

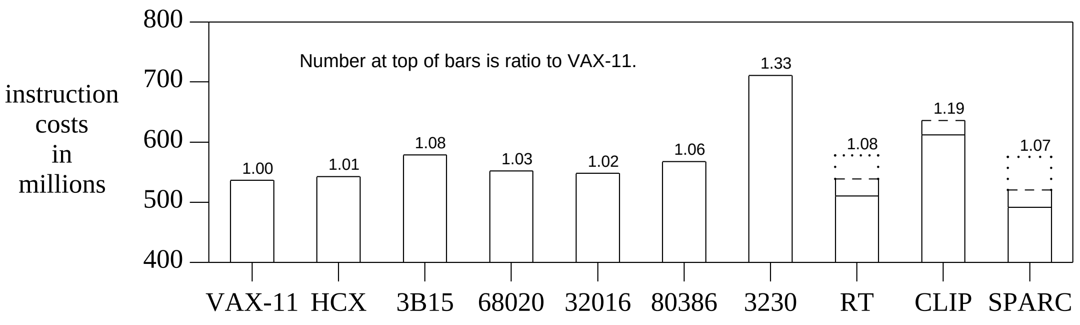


Figure 19. Number of Instruction Cost Units for each Machine

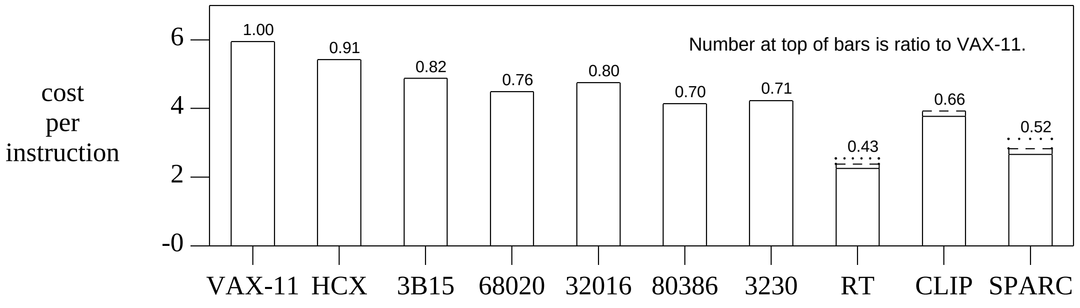


Figure 20: Average Cost per Instruction for each Machine

3.4.3.2. Operand Costs

The operand cost results are shown in Figures 21 and 22. The machines with more complicated addressing modes had a larger cost per operand. Since there were fewer executed instructions and thus fewer operands in the more complicated machines, in general the total operand costs for these machines were smaller. The number of operand costs for the Clipper and SPARC were reduced to due to the large number of available registers on each of these machines. The SPARC had a higher number of operand costs than the Clipper since global addresses had to be constructed using two immediate values. The RT had a large number of operand costs due to its use of a constant pool for generating global addresses.

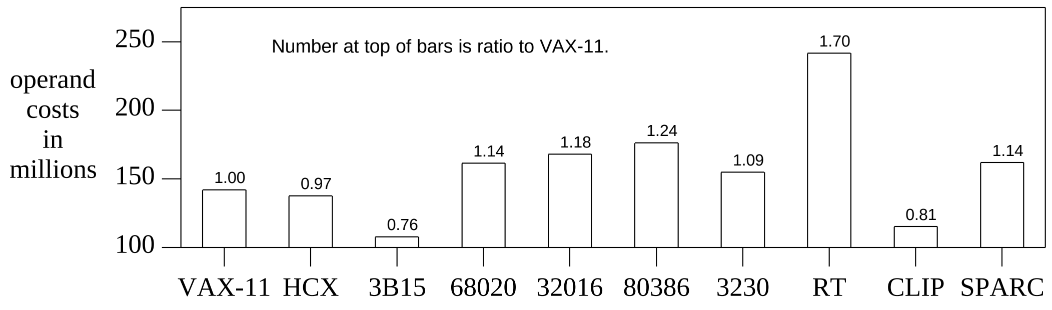


Figure 21: Number of Operand Cost Units for each Machine

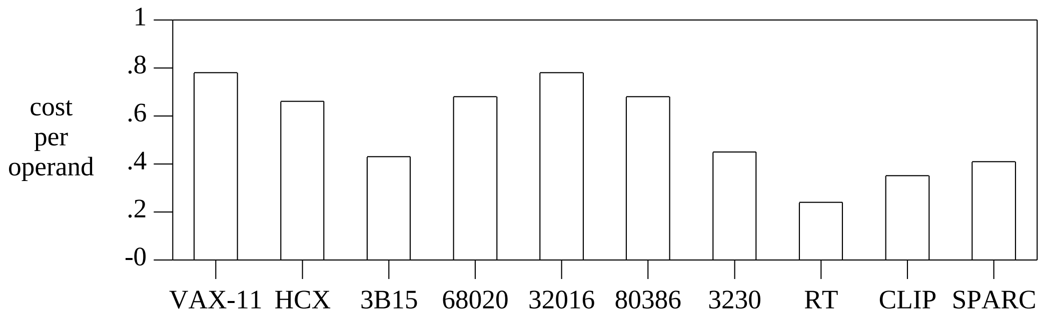


Figure 22: Average Cost per Operand for each Machine

3.4.3.3. Memory Costs

The memory costs were obtained by assigning a cost to each memory reference. In general, RISC machines reference memory less often than CISC machines due to a greater number of available registers. The RT was an exception due to its use of a constant pool for generating global addresses. Each time a global address is needed, the RT loads the desired address from memory. This is shown in Figure

23.

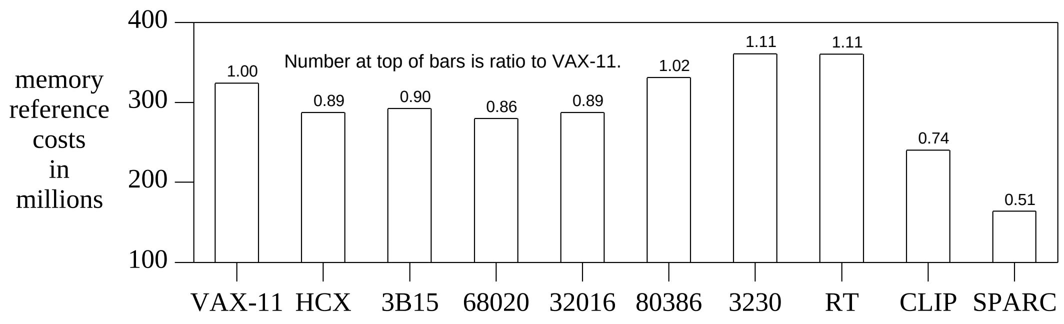


Figure 23: Number of Memory Reference Cost Units for each Machine

3.4.3.4. Total Costs

Figure 24 shows the total generic cost for each machine. The total cost was obtained by adding the instruction, operand, and memory costs.

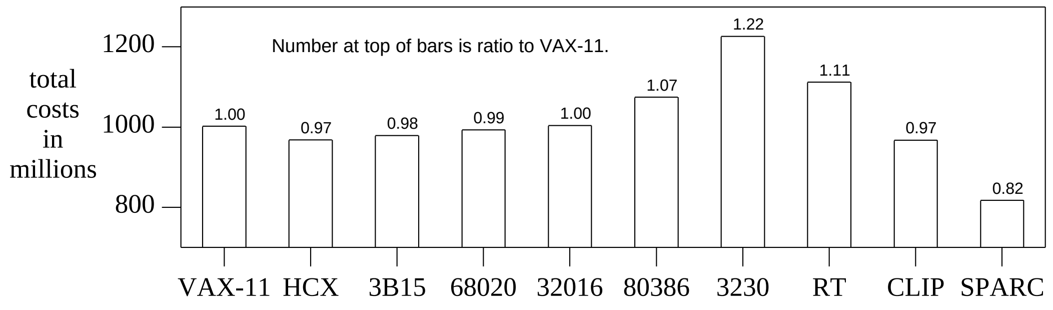


Figure 24: Number of Total Cost Units for each Machine

Without some consideration given to the actual implementation, these results can be misleading. The results seem to imply the machines with the more complicated instructions and addressing modes will perform almost as well as RISC machines. The results are very different if one only considers whether a machine is implemented using microprogramming or not.

Microprogramming a complicated instruction set, such as the VAX-11, has advantages. The more complicated instructions take longer to execute than the simpler instructions. However, to avoid an instruction cycle time as long as required to implement the most time-consuming complex instruction, each macroinstruction can be implemented as a set of microinstructions. This reduces the average execution time for a macroinstruction. Instructions can be implemented more easily and cheaply in a microprogram than in a hardwired control unit. Also the instruction set of a machine can be modified without modifying the underlying hardware.

The main disadvantage to using microprogramming is that it is slower than a hardwired unit. A microcoded control unit typically uses the opcode of the macroinstruction as an address in a table in the control store memory. Each entry in the table is a jump to the beginning of a microroutine. At the end of the microroutine control is returned to execute the next macroinstruction.

The RT, Clipper, and SPARC are machines that can be implemented without microprogramming. Memory is only referenced by load and store instructions. This results in all other operations except for branches being executed in one machine cycle. Since these instructions take approximately the same amount of time, microprogramming would only increase the execution time of the instruction cycle. Since each of these instructions are simple, they can also more easily be hardwired.

Figures 25 and 26 show the adjusted instruction costs and total costs assuming all the machines except for the RT, Clipper, and SPARC are microprogrammed. The assumption is that each instruction on the RISC machines is less expensive since there is no requirement to branch to the microroutine and return from it. Therefore a cost of 1 unit will be subtracted from each instruction executed on these three machines.

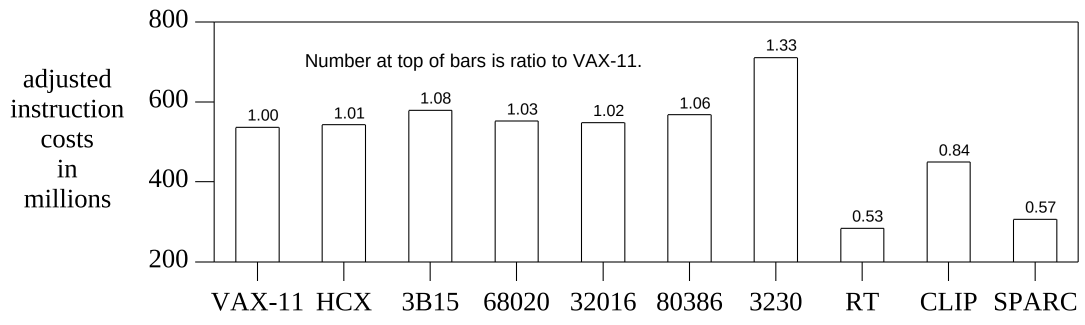


Figure 25: Number of Adjusted Instruction Cost Units for each Machine

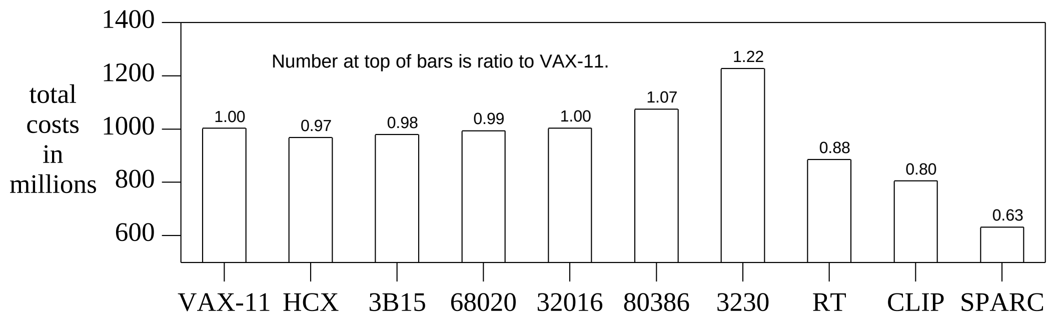


Figure 26: Number of Adjusted Total Cost Units for each Machine

3.4.3.5. Conclusions

These results should be viewed as very rough estimations. If one accepts the stated assumptions, the results from estimating execution costs seem to imply that the RISC machines are more efficient. It appears that the greater number of instructions executed by the RISCs is more than compensated by effective pipelining of loads and branches. The ability for an architecture to be implemented effectively without microprogramming also reduced the execution costs of the RISC machines. Since RISC machines are simpler than CISCs and should be in general easier to design and build, a RISC approach to new architectures may be advisable.

Other factors may be included when comparing each architecture. For instance, the static or dynamic number of bytes of instructions could be included. A machine supporting different size instructions, however, increases instruction decoding time, lengthens the instruction pipeline, and complicates paging algorithms [Hen84, Pat85]. Machines with complex instructions may result in more chip area being used. The design and implementation costs of the hardware supporting architectural features could also be considered. It is very difficult to determine the appropriate weightings for each of these factors.

CHAPTER 4

EXPERIMENTS IN CODE GENERATION AND MACHINE DESIGN

Many experiments can easily be performed using the environment that has been developed as part of this dissertation. One area that was examined is methods for saving and restoring registers. Some of the questions addressed are:

(1) What is the effect of using a caller-save calling sequence versus a callee-save calling sequence?

(2) How many scratch registers should a machine with a callee-save calling sequence have?

(3) How much of a performance gain is obtained by using data flow analysis to minimize the number of saves and restores?

(4) Would a combination of a caller-save and callee-save calling sequences be more effective?

Other aspects of the calling sequence can be examined. After noting that more than 6.5% of memory references found on the VAX were caused by pushing argument values onto the run-time stack, perhaps other schemes for passing arguments may prove to be more efficient. The architectures in the study implement calling sequences with instructions that vary in complexity. Some machines, such as the VAX-11 and HCX-9, use very complex call and return instructions. Other machines, such as the Clipper, use simpler instructions. Some questions that were investigated are:

(1) Is it beneficial to pass arguments in registers without interprocedural analysis?

(2) How often are each of the functions performed by complex call and return instructions necessary?

(3) Would the use of more primitive instructions result in more effective code?

(4) Can additional optimizations be accomplished with primitive call and return instructions?

Branches have been found to be expensive on all of the architectures in the study. One reason for this is that the address calculation and fetching of the branch target is accomplished at the point that the transfer of control is to occur. Therefore, the following ideas were explored.

(1) Is it feasible to separate the address calculation and fetching of the branch target from the transfer of control?

(2) By exposing such a separation to the compiler, can additional optimizations occur?

(3) Would such a separation reduce pipeline delays?

This chapter consists of four experiments. The first is an evaluation of methods to save and restore registers. The next experiment measures the effectiveness of passing arguments through registers as a calling sequence convention. This is followed by an experiment using primitive call and return instructions in combination with passing arguments through registers. The last experiment evaluates a new technique for reducing the cost of branches by using registers.

4.1. Methods for Saving and Restoring Registers

There is a sequence of actions that is required to implement a function call. This sequence of actions, or *calling convention*, is an important aspect of both machine design and programming language implementation. Poor architectural support for subprogram calls, or poor design of the calling convention, can result in slower programs and may lead programmers to avoid the subprogram abstraction mechanism.

An important component of a calling convention is how to preserve the values of the registers that may be used across function calls. Without link-time optimizations [Wal86], interprocedural register allocation [Cho88] or special hardware, such as register windows [PaS82] or dynamic masks [HuL86], there are two methods commonly used to save and restore the values in registers that may be destroyed by the called function. One method delegates the responsibility for saving and restoring the values of the registers to the called function. Upon entry to the called function, the values of the registers that are used and need to be preserved are stored. The values of these registers are then loaded upon exiting the called function. For languages that support recursion, the register values are usually saved in and restored from the activation record of the called function. Since the responsibility of saving and restoring the register values is delegated to the called or callee function, this method is known as *callee-save*.

The other method delegates the responsibility for saving and restoring the registers to the calling function. The calling function stores the values of the registers on the run-time stack before a call instruction. After the call instruction, the function loads the values from the run-time stack into the registers. Since the responsibility of saving and restoring the register values is assigned to the calling or caller function, this method is known as *caller-save*.

There are many factors to consider when choosing a method to save and restore the values in registers across function calls. Choosing a method often involves trading off implementation simplicity for run-time efficiency. To a large extent, design decisions are driven by the architectural support supplied by the target machine and the intuition (bias) of the implementor. This section describes the results of a set of controlled experiments which were used to evaluate several methods for saving and restoring registers [DaW91]. Our experiments show that a hybrid approach, a combination of callee and caller methods, produces the most effective code.

4.1.1. Related Work

There has been some previous work studying methods for saving and restoring registers. In seeking to show that their register window mechanism would efficiently support high-level languages, Patterson and Sequin measured how deeply calls are nested at run-time [PaS82]. They found that the average calling depth during the execution of a program was not very great. Their hardware approach, register windows, avoids saving and restoring registers by switching to a new window at each call. Saves and restores are required only when no more windows are available. With eight register windows, in our investigation less than one percent of the calls and returns required a window to be stored and loaded from memory.

Lang and Huguet [HuL86] analyzed the effectiveness of a simple caller-save method, a simple callee-save method, and six other schemes involving a dynamic mask. Each bit set in a single dynamic mask indicates a register whose value needs to be retained by functions currently active. A static mask, associated with each function, indicates the set of registers it uses. The set of registers saved and restored is computed by taking the intersection of the dynamic and static masks. For each of these schemes, variables were preallocated to registers. They found that the dynamic mask could reduce the total number of saves and restores. They also found that the dynamic mask schemes benefited from a larger set of registers by assigning registers in a round-robin fashion as functions were being compiled.

Chow [Cho88] investigated linking program units together in a compilation environment to allow interprocedural register allocation. He divided the registers of a machine into caller and callee-save sets. Using a technique called shrink-wrapping, the saves and restores for callee-saved registers were placed only around regions where the registers were used. By processing the functions in a depth-first ordering of the call graph, the interprocedural register allocator avoided using registers that could be active at the same time in other functions. It also was used to pass arguments through registers. With a sufficient number of registers, Chow found that the cost of saving and restoring registers at procedure calls could be significantly reduced.

4.1.2. Methods Investigated

The technique used to save and restore register values across function calls can affect performance and influence the design of the instruction set of a machine. The advantages and disadvantages of the possible implementations are not obvious. To better understand the tradeoffs, six possible implementations for saving and restoring register values across function calls were examined. These six methods will be referred to as:

1. simple callee
2. simple caller
3. simple hybrid
4. smarter callee
5. smarter caller
6. smarter hybrid

The three simple methods do not use data flow analysis to minimize the number of saves and restores while the smarter methods do.

4.1.2.1. Simple Callee

The simple callee method, due to its simplicity, is probably the most common method used for saving and restoring registers. The set of allocable registers is broken into two groups, scratch and non-scratch. Local variables and arguments may be allocated to non-scratch registers. Scratch registers are not guaranteed to retain their values across function calls. Consequently, they are only used to hold temporary values, to compute intermediate results, and to return values from the called function to the calling function. Only the non-scratch registers that are used in the function are saved and restored upon function entry and exit respectively.

This is the calling sequence generated by *pcc*-based C compilers [Joh79] on a VAX-11 running 4.3BSD Unix and on a SUN-3 running SunOS 4.0. For both of these compilers, a register declaration of a local variable or argument results in that variable being allocated to a non-scratch register if one is available. On the VAX-11, a mask with a bit set for each register to be saved is stored at the beginning of each function. The `calls` instruction scans the mask pushing the contents of the indicated registers onto the run-time stack. On function exit, the `ret` instruction restores the values into the appropriate registers. On the SUN-3, a Motorola 68020-based machine, special instructions are used to save and restore the non-scratch registers referenced by the called function. These instructions also use a bit mask to specify the registers to save and restore.

4.1.2.2. Smarter Callee

The smarter callee method is similar to the simple callee method with one difference. Instead of placing the saves and restores of non-scratch registers at the function entry and exit respectively, saves and restores are placed only around the region where the registers are used. For instance, if the lives of a non-scratch register are contained within a conditional, such as an if statement, then the save and restore of that register are also placed within the code generated for the if statement. If the execution path when the function is invoked does not enter this region of code, then the save and restore are not performed unnecessarily. Saves and restores, however, are never placed within loops.

4.1.2.3. Simple Caller

The simple caller method places local variables and arguments in any allocable register. With this approach there is no notion of partitioning the available registers into scratch and non-scratch sets. The life of every available register does not extend across function calls and thus all registers are scratch.

Any register that is used to hold a local variable or argument is saved immediately preceding a call and restored immediately following a call.

4.1.2.4. Smarter Caller

The smarter caller method uses data-flow analysis to minimize the number of saves and restores. This analysis allows two different optimizations to be performed. The first optimization eliminates saves and restores that are unnecessary. The second optimization attempts to move necessary saves and restores out of loops. These optimizations are explained in the section that describes the implementation of this approach.

4.1.2.5. Simple Hybrid

How a variable to be allocated to a register can be saved and restored most efficiently depends on how the variable is used within a function. In some cases, it would be cheaper to allocate a variable to a callee-save register to save and restore the register as in a callee-save convention. This occurs when the life of the variable overlaps with call instructions. In other cases, it would be cheaper to allocate a variable to a caller-save register. This occurs when the life of the variable does not overlap with call instructions. Often both types of variables exist within a single function.

The simple hybrid method is a combination of the simple callee and simple caller methods. The set of available registers is divided into two classes. One class, non-scratch registers, is saved and restored by the simple callee method. The cost of saving and restoring a non-scratch register is always two memory references. The other class, scratch registers, is saved and restored by the simple caller method. The save/restore cost of a scratch register is twice the number of memory references as estimated calls executed within the function. The compiler allocates variables to the class with an available register which has the least estimated save and restore cost.

4.1.2.6. Smarter Hybrid

The smarter hybrid method is similar to the simple hybrid method. The difference is that this method is a combination of the smarter callee and smarter caller methods. The set of available registers is partitioned into one set of non-scratch registers that is saved and restored by the smarter callee method and a second set of scratch registers that is saved and restored by the smarter caller method.

4.1.3. Environment for Experimentation

For these experiments, we measured the number of instructions executed, the number of memory references, and the size of the object code. The experiments were performed on a VAX-11 and a Motorola 68020. On both machines, C compilers were constructed for each of the six methods. The only calling convention changes made to the run-time library were those required to implement the new save and restore methods. Other conventions, such as passing arguments on the stack instead of in registers, were not altered.

The number of instructions executed is affected by two factors. Typically, as more variables are allocated to registers, the number of instructions used for saving and restoring registers increases. On the other hand, as frequently used variables are allocated to registers, the number of instructions aside from those used for saving and restoring registers decreases. This occurs since *vpo* reattempts code selection after register allocation and often more optimizations are possible once a variable has been allocated to a register. For instance, the VAX-11 and Motorola 68020 auto-increment and auto-decrement addressing modes are only realizable if a variable is allocated to a register. Use of these addressing modes results in fewer instructions being executed.

The number of dynamic memory references is also affected by the number of variables allocated to registers. As more variables are allocated to registers, the number of memory references for saving and restoring registers increases. Conversely, the number of other types of memory references, loading and storing values from variables, decreases.

The test set used for both architectures contained the same nineteen programs described for the architectural study. For each method, all source files of each program were recompiled on the VAX-11. To more accurately determine the impact of each method, the source files from the C run-time library were also recompiled. The test set comprised a total of 250 source files (including files from the C library). The C library on the SUN-3 was not used in data collection since the sources for the library routines were not available. Thus, the test set used for the Motorola 68020 was the same nineteen programs consisting of only forty-five source files. Data was collected from each of the files compiled by *vpo*. Since the C library was not used for data collection on the Motorola 68020, the results for the VAX-11 more accurately reflect the effectiveness of each method of saving and restoring registers.

4.1.4. Implementation

For each method, the *vpo* compiler attempts to allocate local variables and arguments to registers if expected benefits outweigh the save/restore cost. The benefits are determined by estimating the number of times that a variable will be referenced each time the function is invoked. References inside of loops are weighted by the loop nesting level. The following subsections describe the implementation of each method.

4.1.4.1. Simple Callee

In the simple callee scheme, a local variable is allocated to an available non-scratch register if the compiler estimates that the variable will be referenced at least three times. Similarly, an argument is allocated to an available non-scratch register if the compiler estimates that the argument will be referenced at least four times. An additional reference is required for arguments due to the initial load of the argument from the run-time stack to the register.

4.1.4.2. Smarter Callee

For the smarter callee approach, local variables and arguments are allocated to registers in the same manner as the simple callee method. The only difference between the two methods is the placement of the saves and restores. The placement of these instructions were determined by using the data flow equations for a method called shrink-wrapping described in Cho88.

4.1.4.3. Simple Caller

Unlike the simple callee method, the number of saves and restores in the simple caller method varies depending upon the estimated number of calls made by the function. In the simple caller method a local variable is allocated to an available register if *vpo* estimates that placing the variable in the register will result in fewer overall memory references. This only occurs if the number of estimated references to the variable is greater than the estimated number of saves and restores (twice the estimated number of calls made). An argument is allocated to an available register if the number of references is estimated to be greater than the number of saves and restores plus one. As in the simple callee method an additional reference is required for arguments due to the initial load of the argument from the run-time stack to the register.

4.1.4.4. Smarter Caller

The estimated number of saves and restores of registers associated with an argument or local variable for the smarter caller method is determined in the algorithm described below. Each step in the algorithm refers to one or more examples of specific program flow graph situations, illustrated in Figures 25 through 30. Each arrow between instructions in these figures represent a sequence of instructions that can be executed within the function.

for each register at each call site in a function

(1) Determine if there is a potential *use* of the register after the call. If there is no *use* (only *sets* or returns as shown in Figure 27), then there is no need to save and restore the register associated with the call.

(2) Determine if there is a *use* of the register that follows the current call with no intervening call. If not (only *sets*, calls, and returns as shown in Figure 28), there is no need to restore the register after the current call since it will be restored after a following call.

(3) Determine the cost of a restore before each *use* that follows the current call and is not preceded by another call. If these restores cost less than a restore following the current call, then place restores before each of these *uses*. Otherwise, place a restore immediately after the current call. These choices are illustrated in Figure 29. If there is a *use* of a register that can follow the current call and the *use* can also be reached without the current call being executed, then the restore is always placed after the current call. This is illustrated in Figure 30.

(4) Determine if there is a *set* of the register that precedes the current call with no intervening call. If not (only calls or function entry as shown in Figure 31), there is no need to save the register.

(5) Determine the cost of a save after each *set* that precedes the current call with no intervening call. Determine the cost of a restore after each call that precedes the current call with no intervening *set* of the register. If these saves following each *set* cost less than a save preceding the current call and the restores following preceding calls, then place saves after each of these *sets*. Otherwise, place a save before the current call and a restore following the preceding calls. This choice is shown in Figure 32.

This algorithm is first used to order the arguments and local variables from the greatest estimated benefit to the smallest estimated benefit from allocating the variable to a register. After variables have been allocated to registers and all optimizations have occurred, the algorithm is used to insert the appropriate saves

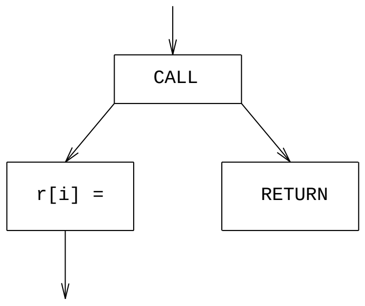


Figure 27. No Use of Register after Call

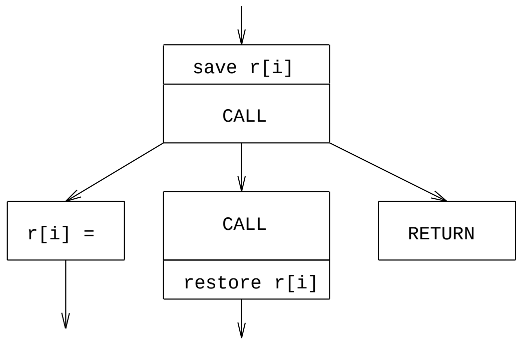


Figure 28. Only Sets, Calls, and Returns after Call

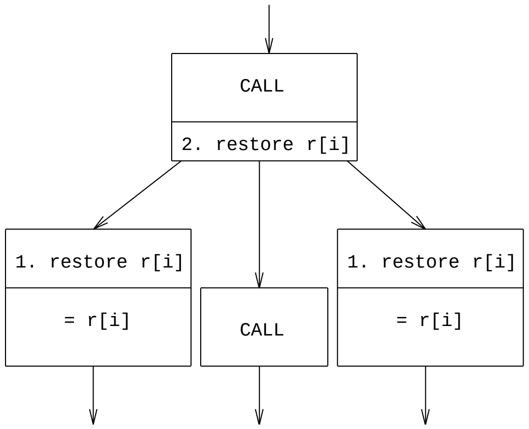


Figure 29. Restores before Uses (1)
or after Call (2)

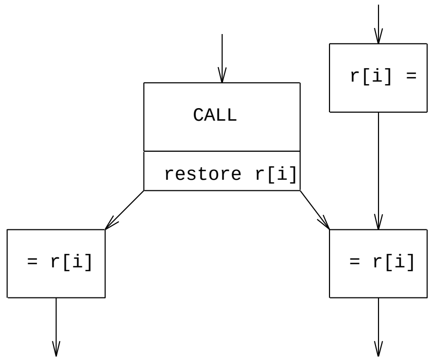


Figure 30. Use Preceded by a Set

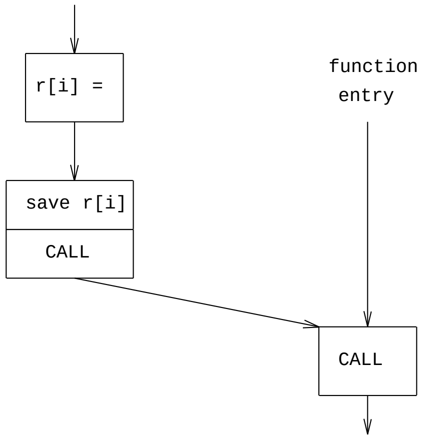


Figure 31. No Sets before Call

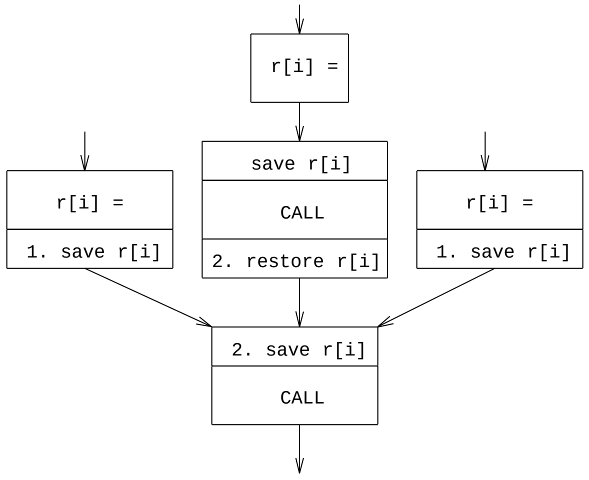


Figure 32. Saves after Sets (1) or before Call (2)

and restores. Thus, the smarter caller method reduces the cost of saving and restoring registers in two ways. First, it saves and restores a register only when necessary. Second, it moves the saves and restores to follow the previous *sets* and to precede the subsequent *uses* if it is beneficial. This has the effect of moving saves and restores out of loops.

4.1.4.5. Simple Hybrid

For the simple hybrid method, *vpo* calculates the number of saves and restores as in both the simple callee and the simple caller methods. When attempting to allocate a local variable or argument to a register, if a register from each set is available, then the costs of saving or restoring the register by the two methods are compared. If it is beneficial to use the register, the method resulting in the least cost is used.

4.1.4.6. Smarter Hybrid

The smarter hybrid approach is similar to the simple hybrid approach. In this approach *vpo* calculates the number of saves and restores as in both the smarter callee and the smarter caller methods and uses the method that has the greatest benefit for allocating a local variable or argument to a register.

4.1.5. Partitioning Registers

Before one can evaluate the effectiveness of a callee-save or hybrid calling sequence, the allocable registers must be divided into two sets. One set of registers, designated as non-scratch, are guaranteed to retain their values across calls, and the other set of registers, designated as scratch, are not. If the set of allocable registers in a callee-save calling sequence is partitioned such that there are too many scratch registers, then not enough variables can be allocated to registers. If there are too many non-scratch registers, then saves and restores of non-scratch registers used only as temporaries will be required. An inappropriate partitioning in a hybrid calling sequence can also result in poorer performance due to fewer variables allocated to registers.

There has been little attention given to determining the best allocation of scratch and non-scratch registers in a callee-save calling sequence. For instance, the fixed-point portions of the Digital Equipment Corporation’s VAX-11 and the Harris Corporation’s HCX-9 architectures are very similar. Yet the

VAX-11 4.3 BSD C run-time library was implemented with six of its twelve user-allocable registers as scratch while the HCX-9 C run-time library was implemented with only two of its thirteen user-allocable registers as scratch. It appears that six registers were designated as scratch on the VAX-11 since the `movc` instruction is hardwired to use `r0` through `r5`.

The callee and hybrid methods involve partitioning the set of available registers into two classes. For these methods, the number of scratch registers was varied to determine the most effective combination of scratch and non-scratch registers. It was found on the VAX-11 that the optimal number of scratch registers from the twelve allocable general-purpose registers was five for the simple callee method, six for the smarter callee method, and eight for the hybrid method. The Motorola 68020 has eight allocable data registers, and six allocable address registers. The most effective number of scratch registers for this machine was discovered to be two data and two address for the simple callee method, three data and two address for the smarter callee method, and four data and three address for the hybrid method. The default number of scratch registers for each register type on this machine was two. Even though the C library on the SUN-3 could not be recompiled, the number of scratch registers can be increased.

To determine if the most effective percentages of allocable registers designated as scratch for twelve registers would be best even if the number of registers available changed, *vpo* for the VAX-11 was modified to produce code assuming that the machine had four, eight, and sixteen user-allocable registers. Modifying the compiler to produce code with four or eight registers simply required specifying the reduced number of user-allocable registers to the register allocator. Having more registers than actually exist required slightly more work. The nonexistent registers were associated with a set of local variables in each function. If a nonexistent register was referenced directly as an operand, then the corresponding local variable was used. If an instruction referenced a nonexistent register that was part of a more complex addressing mode, then the corresponding local variable was loaded into an available register preceding the instruction and the available register was referenced in the instruction instead. If there was a side effect of referencing the register, the autoincrement and autodecrement addressing modes, then additional instructions were generated to update the corresponding local variable. Six existing registers, the maximum number of registers that could be referenced in a single instruction for the

VAX-11 *vpo* compiler, were also associated with a set of local variables. This ensured that there was always an available register in which one could load a local variable that corresponded to a nonexistent register.

Table 24 shows the results of varying both the number of user-allocable registers and the number of scratch registers for the simple callee, simple hybrid, and smarter hybrid methods. The smarter callee results are not shown since the simple callee results were approximately the same. Note that at least two scratch registers are required on the VAX-11 to return a double-precision floating-point value. These results imply that the number of user-allocable registers has little effect on the most effective percentage of scratch registers (about 40% for simple callee, 50% for simple hybrid, and 75% for the smarter hybrid). Only for the smarter hybrid with four registers did the percentage vary significantly.

method	simple callee		simple hybrid		smarter hybrid	
user-allocable registers	scratch registers	memory references	scratch registers	memory references	scratch registers	memory references
4	2	98458465	2	96064606	3	93403276
	3	111117829	3	96354469	4	91751729
8	2	78974787	3	76202403	5	73310177
	3	78846670	4	75810477	6	72841135
	4	79774990	5	76581581	7	73308924
12	4	75512463	5	71894964	7	68958314
	5	75099786	6	71754355	8	68880184
	6	75752791	7	71968620	9	68987123
16	5	72855408	6	69337708	11	65653992
	6	72841124	7	69229771	12	65139854
	7	72924722	8	69302664	13	65593297

Table 24: Results of Scratch/Non-Scratch Combinations

Many callee-save calling sequences seem to be designed using the idea that only a couple of scratch registers are necessary. In several ports of *pcc* [Joh79], only two scratch registers are used even when additional scratch registers are available. There are instances on some machines where more than two scratch registers are required for calculations to avoid spills. Special instructions that require a number of registers, such as the move character instruction on the VAX-11, impact the number of desirable scratch registers. When no special instructions or calls that can update scratch registers are detected within a function, local variables and arguments can be allocated to scratch registers. Since over half the functions entered in the test set met this criteria, this also indicates the need for more scratch registers. By measuring the effect of varying the number of scratch registers, the appropriate number may be determined.

4.1.6. Comparison of the Different Methods

Figures 33 through 38 show the results of using the six different methods on the two machines. For the methods requiring the registers to be partitioned into scratch and non-scratch registers, the best combination as previously determined in the section on partitioning registers was used.

The type of instruction or mechanism available in the architecture to save and restore registers may bias the results for a particular method. For instance, it would be an unfair comparison if the mask associated with the VAX-11 `calls` instruction was used for simple callee method and a comparable strategy for the simple caller method was not employed. To illustrate the effects of the mechanism used to save and restore registers on each of the methods, results are presented in three different ways. The number above each bar is the ratio relative to the simple callee approach for that type of measurement. When there was not room to place the ratio above the bar, it was placed below the bar.

The solid lines represent measurements assuming that the saves and restores are not accomplished by separate instructions. Thus, for a callee-save method a mask is associated with each routine to save and restore non-scratch registers used in a function as is done with the `calls` instruction on the VAX-11. In a caller-save method it was assumed that a mask would be associated with each call instruction to save and restore scratch registers that are assigned to local variables and arguments. Since saves and restores in the smarter methods may be desired at locations other than the function entry, function exit, and calls, this measurement is not appropriate for these three methods. Though use of these masks decreases the number of instructions, the number of memory references is increased since two memory references to save and restore the mask would be required for each function entered. For the simple hybrid method, the caller mask is ored with the callee mask as a routine is invoked and entered, the registers corresponding to those bits saved, and the ored mask also saved. When a return instruction is executed the ored mask is loaded from memory and the set of registers are restored. Two bytes for the mask at the beginning of each function were added for callee-save methods and two bytes for the mask associated with each call instruction were added for caller-save methods.

Results are presented using two other mechanisms to save and restore registers. The dashed lines represent measurements assuming that special instructions are available that can save and restore a set of registers. Measurements indicating that saves and restores are accomplished as separate `mov` instructions are represented with dotted lines. For the three smarter methods it was assumed that these mechanisms would save and restore a specific register always in the same location for an invocation of a function since the set of registers being restored at a given point may not match the last set of registers saved. Since the number of memory references is the same if either special instructions or separate instructions are used to save and restore registers, dotted lines are not shown in the results depicting the number of memory references. A separate instruction to save or restore a register required four bytes on each machine. For each method, these `mov` instructions would reference a memory location reserved on the run-time stack for that register. Special instructions to save or restore a group of registers were assumed to require four bytes on the VAX-11. The `movem` instruction, used to save or restore a set of `a` and `d` registers for the Motorola 68020, required four bytes.

For the VAX-11, Figures 33 through 35 display the total number of instructions executed, the total number of memory references performed, and the static code size. For each measure, the simple callee and smarter callee methods produced similar results. There was only a 0.3% reduction in the number of memory references by using the smarter callee method. The smarter caller method, however, performed better than its corresponding simple implementation. In terms of instructions executed, the simple caller, simple hybrid, smarter caller, and smarter hybrid are roughly equivalent. However, the smarter caller and the hybrid approaches are clearly superior in reducing the number of memory references. It is interesting to note that there was a 4 to 3 ratio of restores to saves in the smarter caller method. More restores may occur when a loop with a call has *uses* and no *sets* of a register. While the restore has to remain in the loop, the save is placed before the head of the loop. Both the smarter caller and hybrid approach reduced the code size by approximately six percent, when individual instructions were used to save and restore each register.

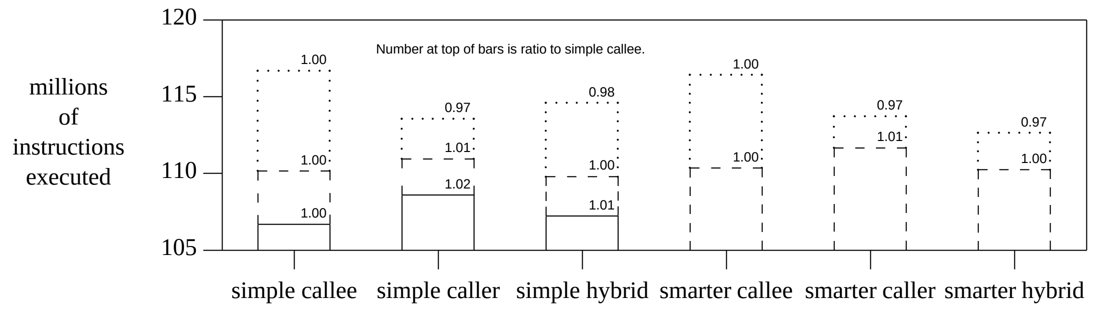


Figure 33: Total Instructions Executed - VAX-11

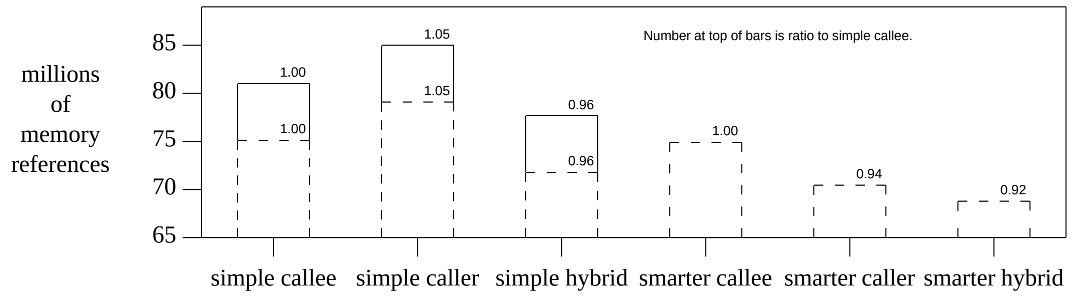


Figure 34: Total Memory References Performed - VAX-11

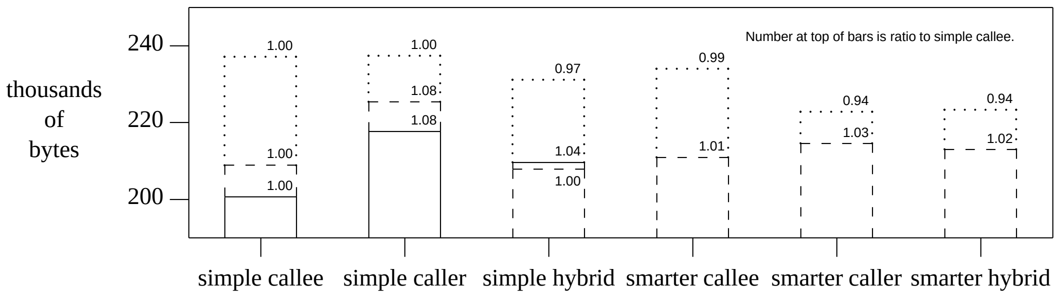


Figure 35: Static Code Size - VAX-11

Figures 36 through 38 display the number of instructions, number of memory references, and static code size for the Motorola 68020. The simple caller method resulted in the most instructions executed and the most memory references. Again the improvement in the smarter callee method over the simple callee method was very slight. Both smarter methods resulted in fewer instructions and memory references than their simpler counterparts. Unlike the VAX-11, the simple hybrid was slightly better than the smarter caller. The poorer performance of the caller methods for the 68020 occurred since the C library on the SUN-3, which contains many leaf functions, could not be used in data collection. The smarter hybrid method had the fewest instructions and memory references and required the least space.

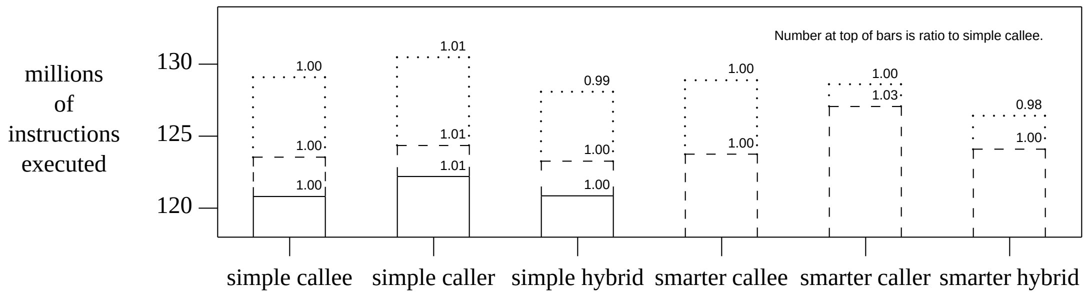


Figure 36: Total Instructions - Motorola 68020

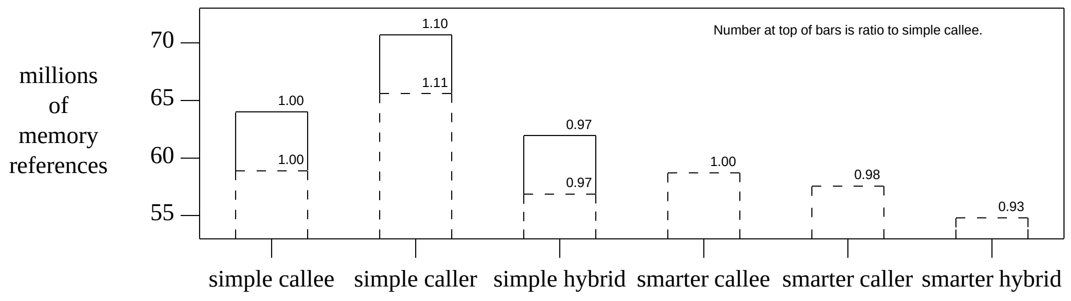


Figure 37: Total Memory References Performed - Motorola 68020

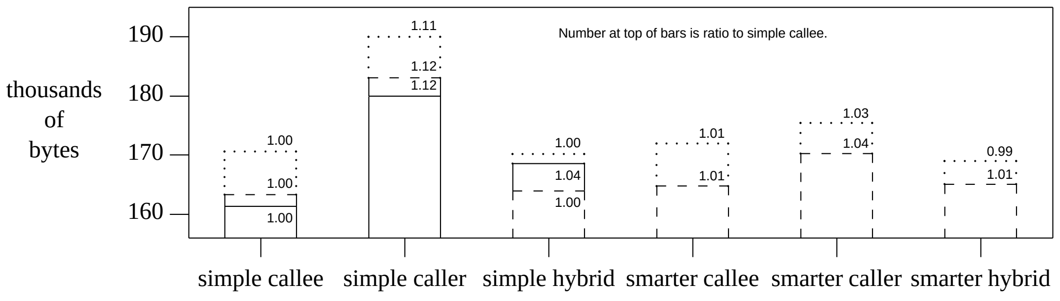


Figure 38: Static Code Size - Motorola 68020

The results for both machines were also affected by the mechanisms in the architecture used to save and restore registers. The reduction in the number of instructions executed from using masks as compared to using separate instructions in the simple methods was achieved at the expense of additional memory references. Special instructions to save and restore a set of registers appear to be an efficient compromise. These special instructions also reduced the number of instructions executed in the smarter methods. This shows that even though dataflow analysis was used to place saves and restores at more beneficial locations in the code, many saves and restores still tended to cluster together.

The availability of hardware support can influence the choice of a method for saving and restoring registers. The VAX-11 `calls` instruction, for example, favors using a callee-save approach. Our experience from implementing compilers for a wide variety of machines has shown that the different methods require different support. For instance, the saving and restoring of registers in the callee-save methods could be implemented more efficiently with instructions that specify a range of registers as opposed to instructions that use a mask to specify the registers to save and restore. This type of instruction should execute faster (no need to scan the mask), and can be encoded in fewer bits than an instruction that uses a bit mask. On the other hand, a caller-save approach favors the use of a bit mask since a contiguous range of registers to save and restore may not occur at each call (indeed it would be rare that it was a contiguous range).

A callee-save method could encode a save/restore instruction without specifying the memory location by referencing memory at the location specified in the stack pointer and automatically updating the stack pointer accordingly. Such an instruction would not be useful in a caller-save method without a frame pointer since arguments are typically placed on the stack at each call.

4.1.7. Future Work

There are still areas involving methods for saving and restoring registers that can be investigated. At the time these experiments were performed, *vpo* allocated local variables and arguments to a single register (or register pair). Techniques known as graph coloring [CAC81, ChH84] can be used to assign registers to the live ranges of variables rather than to the entire variable. Thus, if the live ranges of two variables do not overlap, then both live ranges can be assigned to the same register. Such a technique could reduce the number of saves and restores in a callee-save method since fewer non-scratch registers may be used. There could be benefits in both callee-save and caller-save methods by being able to allocate more variables to registers.

Another area to be investigated is how these methods perform on different machines. We have performed experiments on a VAX-11 and Motorola 68020. Both of these machines are considered Complex Instruction Set Computers (CISCs). It would be interesting to perform these experiments on Reduced Instruction Set Computers (RISCs) as well. For instance, most RISC machines only reference memory by load or store instructions. This would result in an increase in the use of registers as temporaries. Some of these machines also pass arguments through some of the available scratch registers. Again, scratch registers would be used more frequently. Thus, the most effective partitioning of scratch and non-scratch registers in a callee-save or hybrid approach may have a higher number designated as scratch.

A final area to be investigated would be the influence of the type of application or programming language on the effectiveness of these methods. Applications which tend to use more local variables in each function, often found in scientific applications, would perform more efficiently with save/restore methods, such as the hybrid methods, that allow more variables to be allocated to registers. Programs in applications or languages that tend to be more call-intensive may make more use of non-scratch registers.

4.1.8. Conclusions

This study evaluated six different schemes for saving and restoring register values across function calls. Shrink-wrapping, the approach used in the smarter callee method, has been shown to be effective when interprocedural analysis is used [Cho88]. The benefit of using callee dataflow analysis, however, appears to be minimal when optimizations are limited to within the scope of a function. The results from using shrink-wrapping in this paper are comparable to the results reported by Chow for C programs compiled without interprocedural analysis [Cho88]. Therefore, unless some interprocedural optimization is used, such as interprocedural register allocation or inlining, the complexity of implementing shrink-wrapping outweighs the benefits. Caller dataflow analysis, however, was shown to be very effective. For both machines there was over a 10% reduction in the number of memory references by using the smarter caller method instead of the simple caller method.

The hybrid approaches produced better results than using a single method for saving and restoring registers. The results indicate that there are typically some situations where registers can best be saved and restored by a callee method, and other situations where the registers are best handled using a caller save/restore method. The smarter hybrid approach produced the best overall results. Its implementation is only slightly more complicated than the smarter caller approach (if shrink-wrapping is not used) and our measurements showed that it resulted in the fewest number of instructions to be executed and the fewest number of memory references to be performed. It also produced the smallest code on the Motorola 68020. If speed of execution is the most important factor, then the smarter hybrid approach would be the method of choice. The simple hybrid method is an attractive choice if simplicity and compiler speed are the most important factors. While only slightly more complicated than the simple callee and simple caller methods, the simple hybrid approach produces code that is almost as effective as a smarter caller-save approach and it is much simpler to implement. Our production compilers use the simple hybrid approach.

4.2. Parameters Passed in Registers as a Calling Sequence Convention

A significant percentage of memory references when executing programs are due to pushing parameters onto a run-time stack and referencing those parameters from the run-time stack. This section describes an experiment that changes the calling sequence on a machine to allow passing up to six parameters in registers.

4.2.1. Related Work

Passing parameters through registers is not a new idea. In 1978 Tanenbaum discovered from dynamic measurements of 300 procedures in SAL that 72.1% of the calls executed had two or less arguments and 98% were passed in five or less arguments [Tan78]. A hardware approach to this problem is to allow parameters to be passed to routines in registers by use of overlapping sets of registers in register windows [Pat85]. Lisp compilers have been implemented with arguments passed in registers since the early 1980s [GrB82]. There have been several approaches to allow parameter passing through registers involving interprocedural analysis [BeD88, Cho88, Wal86].

4.2.2. Environment for Experimentation

Vpo was modified to collect measurements on the VAX-11. As stated previously, the VAX-11 has twelve user-allocable registers (`r0-r11`) which are used to contain both integer and floating-point values. The calling sequence was modified to pass up to six parameters through the scratch registers. Double-precision floating-point parameters require two registers. At the point of the call, the compiler starts with the last parameter that would be pushed onto the run-time stack and stores it in a scratch register (`r5` if an integer, `r4` if a double). The compiler continues placing parameters in scratch registers until there are no more arguments or available scratch registers. Any parameters that cannot be passed through registers are pushed onto the run-time stack. The registers containing the arguments are stored in memory as a local variable at the function entry. Since the compiler knows the order of the arguments, it can store the correct register in the appropriate variable. This modification does not affect the availability of the scratch registers since they cannot be used across calls.

Using reference estimates, *vpo* allocates the most frequently used variables to registers. Two memory references are saved if the parameter is allocated to a register. The memory references that are eliminated are the pushing of the value on the run-time stack in the calling function and loading the value from the stack into a register in the called function. If the parameter is not allocated to a register, an additional instruction is required over the traditional scheme of using the run-time stack for passing parameters. The number of memory references in this case is unchanged.

A greater number of arguments should be allocated to registers using this scheme. *Vpo* will not allocate a local variable or parameter to a register unless the number of estimated references is greater than the cost of allocating the variable to a register. The save and restore cost of both local variables or parameters allocated to registers is two memory references. An additional cost of one memory reference is estimated for a parameter since the parameter has to be loaded from the stack into a register. Thus, *vpo* requires at least four estimated references of a parameter passed on the run-time stack for the parameter to be allocated to a register. Parameters passed through registers do not require loading from the run-time stack. An additional memory reference will also be found since the register containing the parameter is stored in memory at the function entry. Thus, a parameter passed through a register that is referenced only twice in a function can be allocated to a register by the compiler.

A number of special conditions had to be resolved to implement the new calling sequence. There are a set of routines which could not be compiled by *vpo*, since their source code was not available. These system calls were identified and parameters were always passed to these routines on the run-time stack. The set of routines in the C run-time library that accept a varying number of arguments on the run-time stack (e.g. printf, scanf, etc) were also identified. This set of routines depends on the calling sequence convention of how parameters are passed. For instance, they are dependent on the direction that the run-time stack grows and the order in which arguments are evaluated. Any parameter to these routines that was optional was passed on the run-time stack. Structure arguments are pushed on the run-time stack by *vpo* using `movc` instructions. Thus, structure arguments in the new calling sequence were always passed on the stack.

Data was collected using both the default calling sequence convention of passing arguments on the stack and passing arguments through registers. The C run-time library was also compiled by *vpo* and measurements collected to allow a more accurate estimate of the effect of passing parameters through registers. The test set consisted of nineteen programs and 250 files. This was the same set of test programs used in the architectural study.

4.2.3. Results

Table 25 shows the results of running the test suite using the two calling sequences. The results show that passing arguments through registers can effectively reduce the number of memory references. This simple change to the calling sequence convention resulted in 8.4% fewer memory references. The reduced number of memory references resulted from not having to push arguments onto the run-time stack and referencing those arguments from the run-time stack. This savings was possible due to an increase from an average of 0.57 to 1.51 arguments allocated to registers per function invoked. There was also 1.0% more instructions executed. Use of a link-time optimizer, such as VLINK [BeD88], could remove these extra instructions.

measurement	default	params through regs
instructions	104,886,495	106,616,578
memory references	73,085,389	63,415,387
total memory references	96,148,682	88,103,262
param regs per call	0.57	1.51

Table 25: Results of the Two Calling Sequences

Passing parameters in registers has been recognized as beneficial in the past. There has been a variety of schemes used to implement this feature including register windows and link-time optimizations. Registers windows, though effective, has many disadvantages that include the area required on the chip for the large number of registers, the increase in instruction cycle time due to longer access to a register due to having to go through a register window pointer register, and increased process switching time [Hen84]. Link-time optimization requires fairly complex software and increases the time in the compilation process. By simply changing the calling sequence a significant improvement can be obtained without expensive hardware or software. A few recent RISC machines pass arguments through registers as a calling sequence convention. Some of these machines allow only a subset of the available registers for passing arguments. For instance, the Clipper only allows at most two arguments to be passed through registers. This experiment has shown that all of the scratch registers in a callee-save calling sequence can be used to pass arguments without affecting the availability of these registers.

4.3. Using Simpler Call and Return Instructions

If a machine allows recursion, a run-time stack containing information about the state of each active function typically is maintained. To save space and reduce the total number of instructions for maintaining this information, complex call and return instructions have been used in many architectures. Use of these complex instructions for implementing a calling sequence may result in shorter but more inefficient programs. Many of the executed calls do not require all of the functions performed by these complex instructions. Use of these complex instructions also results in missed optimizations available with more primitive instructions. This section describes an experiment that abandons complex call and return instructions available on a machine in favor of more primitive instructions.

4.3.1. Related Work

There has been much work in the past to attempt to speed up function calls. A variety of instructions are available on different architectures to accomplish function linkage. Register windows have been used as a hardware approach to avoiding access to a run-time stack when the calling depth is shallow [Pat85]. Link-time optimizations have been used to avoid more complex call and return instructions in specific situations [BeD88]. Er [Er83] discussed three different schemes for optimizing function calls when the call is the logical last statement in the calling function. Powell [Pow84] stated that his Modula-2 compiler produced code that executed faster when a more expensive procedure call mechanism was replaced with a simpler, faster one.

4.3.2. Primitive Instructions Used to Implement Calls and Returns

The VAX-11 `calls` and `ret` instructions save and restore the number of longwords of arguments, the register mask, the program counter, the frame pointer, and the argument pointer. The number of longwords of arguments is used by the `ret` instruction to adjust the stack pointer back to the value prior to the caller pushing arguments onto the stack. The register mask is read by the `calls` instruction from the first word in the function and is used by the two instructions to save and restore allocable non-scratch registers.

The VAX-11 *vpo* compiler used in the previous experiment for passing arguments through registers was modified to use more primitive call and return instructions. The modified compiler uses the VAX-11 `jsb` and `rsb` instructions which only save and restore the program counter. The `pushr` and `popr` instructions are used to save and restore allocable non-scratch registers, the frame pointer, and the argument pointer. The frame pointer and argument pointer, if used, are adjusted by subtract instructions in the called function. The stack pointer, adjusted by subtract instructions in both schemes at the function entry to allocate space for locals and temporaries, is restored by an add instruction in the new scheme before each `rsb` instruction. These changes are illustrated by the example in Figure 39.

```
        calls   $2,_foo
        ...
.globl _foo
_foo:
.word 0x0FC0
        subl2   $4,sp
        ...
        ret

        =>

        jsb     _foo
        addl2   $8,sp
        ...
.globl _foo
_foo:
        subl2   $4,sp
        pushr   $0x3FC0
        addl3   $32,sp,ap
        addl3   $32,sp,fp
        ...
        popr    $0x3FC0
        addl2   $4,sp
        rsb
```

Figure 39: Primitive Instructions for Implementing Calls and Returns

Fewer data memory references should occur using the more primitive instructions. A large percentage of total memory references can be attributed to handling function linkage. In contrast to the `calls` and `ret` instructions, `pushr` and `popr` only save and restore the frame pointer and argument pointer if they are used. With the six scratch registers being used to pass arguments, very few of the compiled routines will use the argument pointer. This also means that the stack pointer rarely requires adjustment after a `jsb` instruction. The frame pointer need not be adjusted if all locals are allocated to registers. More than 55% of the routines executed in the test set were leaves. Many of these leaves have all of its local variables allocated to scratch registers and thus will not require use of the `pushr` and `popr` instructions for saves and restores.

4.3.3. Optimizations Available

By using the primitive `jsb` and `rsb` instructions in combination with passing arguments through registers, many optimizations on calls can now be accomplished. Because arguments are passed through registers, typically at the point of the call the stack pointer is only adjusted to place the return address on the stack. Optimizations can be accomplished when such a call is immediately followed by an unconditional jump, the sequence of instructions to return to the caller, or another call.

A call followed by an unconditional jump can be optimized to avoid executing the unconditional jump. The `jsb` instruction pushes the address of the next instruction on the stack and jumps to the beginning of the function. To avoid the unconditional jump, the destination of the unconditional jump following the call can be pushed on the stack and then an unconditional jump to the routine can be used. When the called routine executes its `rsb` instruction, control will be transferred to the destination of the original unconditional jump. This is shown by the example in Figure 40.

```
jsb     _foo
jbr     L1

=>

pushl   $L1
jmp     _foo
```

Figure 40: Optimization of a Call Followed by an Unconditional Jump

A call followed by the sequence of instructions to return to the caller can be optimized to avoid the execution of the `rsb` instruction. The `rsb` instruction pops the return address off the run-time stack and branches to that address. To avoid execution of the `rsb` instruction, the sequence of instructions to return to the caller preceding the `rsb` are placed before the `jsb`, the `rsb` is removed, and the `jsb` is replaced by an unconditional jump. This results in the stack pointer being adjusted to point to the return address currently on the stack. Thus, two memory references are also avoided since there is no push and pop of the return address. This is illustrated in Figure 41.

```
jsb     _foo
popr    $0x40C0
addl2   $40,sp
rsb

=>

popr    $0x40C0
addl2   $40,sp
jmp     _foo
```

Figure 41: Optimization of a Call Followed by a Return Sequence

Before attempting this type of optimization for a language such as Pascal, one must ensure that the called routine does not reference a variable declared in the calling routine. If such a variable is contained in the activation record of the calling routine, then its space should not be deallocated from the run-time stack until after the called routine references the variable.

A call to the current function followed by a return is known as tail recursion. The call and the sequence of instructions used to implement the return can be replaced by a branch to the function entry following any instructions used to save the state of the caller. This is illustrated in Figure 42.

Sometimes a sequence of calls occurs with no intervening instructions. Instead of returning to the calling routine after executing each called routine, control can be transferred directly to the beginning of the next routine to be invoked in the sequence. To avoid executing these calls, the address of the instruction following the last call is pushed on the stack, the address of each call except for the first is pushed on the stack in reverse order, and an unconditional jump is made to the first function called. When the first function executes its `rsb` instruction, control will be transferred to the beginning of the second function to be called since its address was on the stack. When the last function executes its `rsb` instruction, control will be transferred to the instruction following the last call. An example of this optimization is shown in Figure 43.

```
.globl _foo
_foo:
       subl2    $4,sp
       pushr    $0x20C0
       addl3    $12,sp,fp
       ...
       movl     r6,r5
       jsb      _foo
       popr     $0x20C0
       addl2    $4,sp
       rsb

       =>

.globl _foo
_foo:
       subl2    $4,sp
       pushr    $0x20C0
       addl3    $12,sp,fp
LB1:
       ...
       movl     r6,r5
       jmp      LB1
```

Figure 42: Optimization of Tail Recursion

```
        jsb      _foo1
        jsb      _foo2
        jsb      _foo3

        =>

        pushl    $L1
        pushl    $_foo3
        pushl    $_foo2
        jmp      _foo1
L1:
```

Figure 43: Optimization of a Sequence of Calls

To enable the three types of optimizations with calls to be performed more frequently, the compiler will attempt to move instructions immediately following a call to before the call. An instruction can only be moved to precede a call if it does not:

1. adjust the stack pointer
2. reference a scratch register
3. reference a global variable
4. reference a variable that has been used indirectly
5. set condition codes that will be subsequently used
6. change the program counter

The same test programs and test data used in the previous experiment of passing arguments through registers was used in this experiment. Data was collected and the results were compared.

4.3.4. Results

Data was collected using the default calling sequence, passing arguments through registers, and passing arguments through registers along with using more primitive call and return instructions. Table 26 compares the results of the three different versions.

measurement	default	params through regs	params through regs and primitive insts
instructions	104,886,495	106,616,578	111,117,985
calls	2,995,675	2,995,675	2,476,404
function linkage memrefs	23,513,272	23,513,272	6,172,169
total memory references	96,148,682	88,103,262	70,762,159

Table 26: Results of the Three Calling Sequences

The results in Table 26 show the benefits of using more primitive call and return instructions. By not performing all of the functions associated with the more complex call and return instructions, there were 19.7% fewer memory references than the version that only passed arguments through registers. There were also 4% more instructions executed. These additional instructions are simpler and less costly than the fewer complex instructions they replaced. Using the new optimizations and movement of instructions before calls, 17.3% (statically 17.6%) of calls were optimized into other instructions. Of these calls that were optimized, 43.4% (statically 45.7%) were followed by a return sequence, 35.4% (statically 30.3%) were followed by an unconditional jump, and 21.1% (statically 24.1%) were followed by another call. Tail recursion optimizations, which have received some attention in the past, was found to occur very infrequently in the test set. Compilation techniques for other paridigms may apply tail recursion more often. For instance, some Scheme compilers translate the source code into an intermediate form, continuation-passing style, where tail recursion is explicit [KKR86].

Maintaining the state of each active function on a run-time stack has been recognized as expensive in the past [Lun77, Wie82]. To address this problem, a variety of complex instructions have been implemented on different machines. It has been shown that many of the functions performed by these complex instructions are unnecessary a large percentage of the time. By using the primitive instructions along with passing arguments through registers, new optimizations were available on 12% of the calls. By moving instructions that follow a call to instead precede the call, when it is possible, it was found that these optimizations were available on over 17% of calls. By simply changing the calling sequence to use more primitive instructions in combination with passing arguments through registers a significant improvement can be obtained. This improvement for a few benchmark programs is shown in Table 27, which compares the execution times of the benchmarks between the original version of the compiler and the version modified to pass parameters through registers and use more primitive call and return instructions. Dhrystone, a call-intensive program, had a significant improvement (the number of dhrystones is shown inside parenthesis). The other benchmarks, which had less frequently occurring call instructions, had a smaller performance gain.

program	params through stack and complex insts	params through regs and primitive insts
dhrystone	8 (5952)	6 (7537)
matmult	0.583	0.567
puzzle	1.650	1.617

Table 27: Comparison of Execution Times

4.4. Reducing the Cost of Branches by Using Registers

Branch instructions cause many problems for machines. Branches occur frequently and thus a large percentage of the machine time is spent branching to different instructions. Branches can result in the pipeline having to be flushed, which reduces its effectiveness and makes pipelines with few stages more attractive. During this delay while the next instruction is to be fetched, no useful work is accomplished on many machines. Since many branch target addresses are a significant distance away from the branch instruction (e.g. calls), the processor will often have greater delays due to cache misses.

A technique is presented that can eliminate much of the cost due to branches by using a new set of registers [DaW90b]. A field is dedicated within each instruction to indicate a branch register containing the address of the next instruction to be executed. Branch target address calculations are accomplished in separate instructions from the instruction causing the transfer of control. By exposing to the compiler the calculation of branch target addresses as separate instructions, the number of executed instructions can be reduced since the calculation of branch target addresses may be moved out of loops. Much of the delay due to pipeline interlocks is eliminated since the instruction at a branch target is prefetched at the point the address is calculated. This prefetching of branch targets can also decrease the penalty for cache misses. The following sections describe this technique in more detail.

4.4.1. Review

Due to the high cost of branches, there has been much work proposing and evaluating approaches to reduce the cost of these instructions. One scheme that has become popular with the advent of RISC machines is the delayed branch. While the machine is fetching the instruction at the branch’s target, the instruction behind the branch is executed. For example, this scheme is used in the Stanford MIPS [HeG83] and Berkley RISC [PaS82] machines. Problems with delayed branches include requiring the compiler or assembler to find an instruction to place behind the branch and the cost of executing the branch itself.

A technique to reduce the cost of executing the branch itself is branch folding. This has been implemented in the CRISP architecture [DiM87a]. Highly encoded instructions are decoded and placed into a wide instruction cache. Each instruction in this cache contains an address of the next instruction to be executed. Unconditional branches are folded into the preceding instruction since the program counter is assigned this new address for each instruction. Conditional branches are handled by having two potential addresses for the next instruction and by inspecting a static prediction bit and the condition code flag to determine which instruction to take. If the setting of the condition code (the compare) is spread far enough apart from the conditional branch, then the correct instruction can be fetched with no pipeline delay. Otherwise, if the incorrect path is chosen, then the pipeline must be flushed. The problems with this scheme include the complex hardware needed to implement the machine and the large size needed for an instruction cache since each decoded instruction is 192 bits in length.

An approach to reduce delays due to cache misses is to prefetch instructions into a buffer [RaR77]. The conditional branch instruction causes problems since one of two target addresses could be used [RiF72]. One scheme involves prefetching instructions both behind the branch and at the target of the branch [LeS84]. This scheme requires more complicated hardware and must also deal with future conditional branch instructions. Other schemes use branch prediction in an attempt to choose the most likely branch target address [LeS84]. If the incorrect path is selected, then the execution must be halted and the pipeline flushed.

4.4.2. Overview of Using the Branch Register Approach

As in Wilke’s proposed microprogrammed control unit [WiS53] and the CRISP architecture [DiM87b], every instruction in the branch register approach is a branch. Each instruction specifies the location of the next instruction to be executed. To accomplish this without greatly increasing the size of instructions, a field within the instruction specifies a register that contains the virtual address of the instruction to execute next.

For instructions specifying that the next instruction to be executed is the next sequential instruction, a branch register is referenced which contains the appropriate address. This register is, in effect, the program counter (PC). While an instruction is being fetched from the instruction cache, the PC is always incremented by the machine to point to the next sequential instruction. If every instruction on this machine is thirty-two bits wide, then this operation can always be performed in a uniform manner. Once an instruction has been fetched, the value of the branch register specified in the instruction is used as an address for the next instruction. At the point the PC is referenced, it will represent the address of the next sequential instruction. An example of this is shown in the RTL below, where `b[0]` (a branch register) has been predefined to be the PC.

```
r[1] = r[1] + 1; b[0] = b[0];  /* go to next sequential instruction  */
```

Since references to `b[0]` do not change the address in `b[0]`, subsequent examples of RTLs will not show this assignment.

If the next instruction to be executed is not the next sequential instruction, then code is generated to calculate and store the virtual address of that instruction in a different branch register and to reference that branch register in the current instruction. Storing the virtual address of a branch target instruction into a branch register on this machine also causes the address to be sent to the instruction cache to prefetch the instruction. The prefetched instruction will be stored into the one of a set of instruction registers that corresponds to the branch register receiving the virtual address and the address in the branch register will be incremented to point to the instruction after the branch target. The instruction register `i[0]`, that corresponds to the branch register `b[0]`, which is used as the program counter, is always loaded with the next sequential instruction.

The first two stages in the pipeline for this machine are the instruction fetch and decode stages. During the decode stage of the current instruction, the bit field specifying one of the branch registers is also used to determine which instruction register to use in the decode stage of the next instruction. When a branch register is referenced in an instruction to indicate that a transfer of control is to occur, the next instruction is executed from the corresponding instruction register. Assuming there is also an execute stage in the pipeline, the dataflow paths between the pipeline stages, registers, and cache are illustrated in Figure 44.

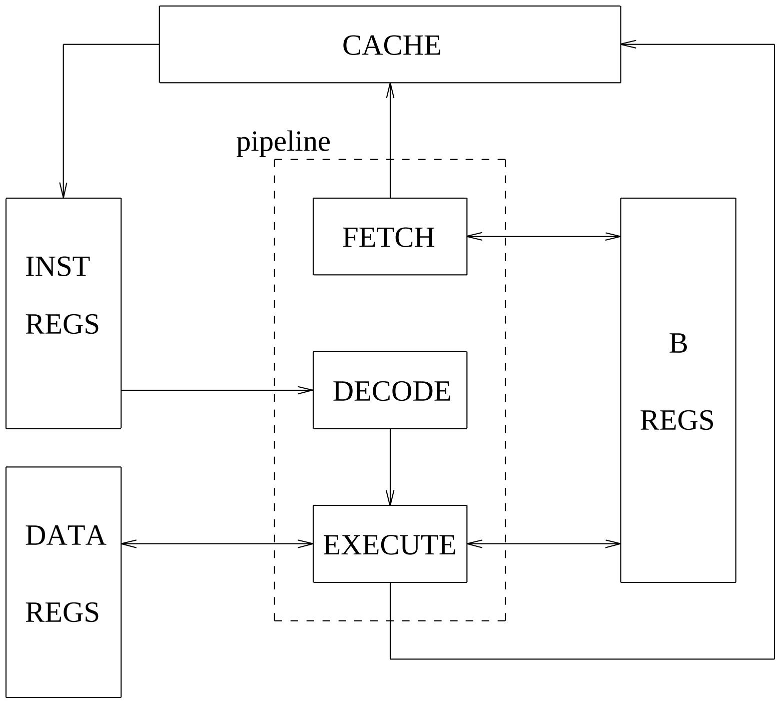


Figure 44: Dataflow for Branch Register Machine

4.4.3. Generating Code for Transfers of Control

The following sections describe how code can be generated to accomplish various transfers of control using branch registers.

4.4.3.1. Calculating Branch Target Addresses

For any instruction where the next instruction to be executed is not the next sequential instruction, a different branch register from the PC must be specified and the virtual address it contains must have been previously calculated. Assuming each instruction on this machine is thirty-two bits wide, a virtual address of thirty-two bits cannot be referenced as a constant in a single instruction. Instead, most instructions could use an offset from the PC to calculate branch addresses. The compiler knows the distance between the PC and the branch target if both are in the same routine. The calculation of a branch target address in a single instruction is shown in the following RTLs:

```
      b[1] = b[0] + (L2 - L1);  /* store address of L2 */
L1:   .
      .
      .
L2:   .
```

For calls or branch targets that are known to be too far away, the calculation of the branch target address requires two instructions. One part of the address is computed by the first instruction and then the other part in the second. Global addresses are calculated in this fashion for programs on the SPARC architecture [SSS87]. An address calculation requiring two instructions is illustrated by the following RTLs:

```
      r[5] = HI(L1);         /* store high part of addr */
      b[1] = r[5] + LO(L1);  /* add low part of addr */
      .
      .
L1:   r[0] = r[0] + 1;       /* inst at branch target */
      .
      .
```

4.4.3.2. Unconditional Branches

Unconditional branches would be handled in the following manner. First, the virtual address of the branch target would be calculated and stored in a branch register. To perform the transfer of control, this branch register would be moved into the PC (`b[0]`), which causes the instruction at the target address to be decoded and executed next. While the instruction at the branch target is being decoded, the instruction sequentially following the branch target is fetched. An example of an unconditional branch is depicted in the following RTLs:

```
      b[2] = b[0] + (L2 - L1);          /* store addr of L2 */
L1:   .
      .
      .
      r[1] = r[1] + 1; b[0] = b[2];     /* next inst at L2 */
      .
      .
L2:   .
```

4.4.3.3. Conditional Branches

Conditional branches would be generated by the following method. First, the virtual address of the branch target is calculated and stored in a branch register. At some point later, an instruction determines if the condition for the branch is true. Three branch registers are used in this instruction. One of two registers is assigned to the destination register depending upon the value of the condition. To more effectively encode this compare instruction, two of the three registers could be implied. For instance, the RTLs in the following example show how a typical conditional branch could be handled. The destination branch register is `b[7]`, which is by convention a trash branch register. The other implied branch register, the source register used when the condition is not true, is `b[0]`, which represents the address of the instruction sequentially following the transfer of control. An instruction following this conditional assignment would reference the destination branch register.

```
      b[2] = b[0] + (L2 - L1);          /* store addr of L2 */
L1:   .
      .
      .
      b[7] = r[5] < 0 -> b[2] | b[0];   /* if cond true then assign b[2] to b[7]
                                           else assign b[0] */
      r[1] = r[1] + 1; b[0] = b[7];     /* next inst at addr in b[7] */
      .
      .
L2:
```

4.4.3.4. Function Calls and Returns

Other transfers of control can also be implemented efficiently with this approach. For example, function calls and returns can be handled by specifying virtual addresses for branch registers. Since the beginning of a called function is an unknown distance from the PC, its virtual address is calculated in two instructions and stored in a branch register. Then, an instruction at some point following this calculation would reference that branch register. To accomplish a return from a function, the address of the instruction following the call would be stored in an agreed-on branch register (for example `b[7]`). Every instruction that references a branch register that is not the program counter, `b[0]`, would store the address of the next physical instruction into `b[7]`. If the called routine has any branches other than a return, then `b[7]` would need to be saved and restored. When returning to the caller is desired, the branch register is restored (if necessary) and referenced in an instruction. An example that illustrates a call and a return on this machine is given in the following RTLs.

```
        r[2] = HI(_routine);                         /* store high part of addr */
        b[3] = r[2] + LO(_routine);                  /* add low part of addr */
        .
        .
        r[0] = r[0] + 1; b[0] = b[3]; b[7] = b[0];   /* next inst is first inst in routine */
        .
        .
   _routine:
        .
        .
        r[0] = r[12]; b[0] = b[7];                   /* return to caller */
```

4.4.3.5. Indirect Jumps

For implementation of indirect jumps, the virtual address could be loaded from memory into a branch register and then referenced in a subsequent instruction. The following RTLs show how code could be accomplished for a switch statement.

```
        r[2] = r[2] << 2;                  /* setup r2 as index in table */
        r[1] = HI(L01);                    /* store high part of L01 */
        r[1] = r[1] + LO(L01);             /* add low part of L01 */
        b[3] = L[r[1] + r[2]];             /* load addr of switch case */
        .
        .
        r[0] = r[0] + 1; b[0] = b[3];      /* next inst is at switch case */
   L01: .long Ldst1
        .long Ldst2
        .
        .
```

4.4.4. Compiler Optimizations with Branch Registers

Initially, it may seem there is no advantage to the method of handling branches on this machine. Indeed, it appears more expensive since an instruction is required to calculate the branch target address and a set of bits to reference a branch register is sacrificed from each instruction to accomplish transfers of control. However, one only needs to consider that the branch target address for unconditional jumps, conditional jumps, and calls are constants. Therefore, the assignment of these addresses to branch registers can be moved out of loops. Since the transfers of control occur during the execution of other instructions, the cost of these branches after the first iteration of loops disappears.

Since there is a limited number of available branch registers, often not every branch target can be allocated to a unique branch register. Therefore, the branch targets are first ordered by estimating the frequency of the execution of the branches to these targets. The estimated frequency of execution of each branch is used, rather than the execution of each branch target instruction, since it is the calculation of the virtual address used by each branch that has the potential for being moved out of loops. If there is more than one branch to the same branch target, then the estimated frequency of each of these branches are added together.

First, the compiler attempts to move the calculation of the branch target with the highest estimated frequency to the preheader of the innermost loop in which the branch occurs. The preheader is the basic block that precedes the first basic block that is executed in the loop (or the head of the loop). At this point the compiler tries to allocate the calculation of the branch target address to a branch register. If the loop contains calls, then a non-scratch branch register must be used. If a branch register is only associated with branches in other loops that do not overlap with the execution of the current loop, then the branch target calculation for the branch in the current loop can be allocated to the same branch register. If the calculation for a branch target can be allocated to a branch register, then the calculation is associated with that branch register and the preheader of that loop (rather than the basic block containing the transfer of control) and the estimated frequency of the branch target is reduced to the frequency of the preheader of the loop. Next, the calculation of the branch target with the current highest frequency is then attempted to be moved out of its innermost loop. This process continues until all branch target calculations have been moved out of loops or no more branch registers can be allocated.

To further reduce the number of instructions executed, the compiler attempts to replace no-operation (noop) instructions, that occur when no other instruction can be used at the point of a transfer of control, with branch target address calculations. These noop instructions are employed most often after compare instructions. Since there are no dependencies between branch target address calculations and other types of instructions that are not used for transfers of control, noop instructions can often be replaced.

Figures 45 through 47 illustrate these compiler optimizations. Figure 45 contains a C function. Figure 46 shows the RTLs produced for the C function for a conventional RISC machine with a delayed branch. Figure 47 shows the RTLs produced for the C function for a machine with branch registers. In order to make the RTLs easier to read, assignments to `b[0]` that are not transfers of control and updates to `b[7]` at instructions that are transfers of control are not shown. The machine with branch registers had one less instruction due to a noop being replaced with a branch target address calculation. Since branch target address calculations were moved out of loops, there were only eleven instructions inside of loops for the branch register machine as opposed to sixteen for the machine with a delayed branch.

```
int foo(a)
int a;
{
   int i, j, k;

   j = 0; k = 0;
   for (i = 0; i < 5; i++)
      if (a < i)
         j++;
   if (j == 5)
      for (i = 0; i < 10; i++) {
         if (a < -5)
            k += 6;
         else
            k += 7;
         }
  return k;
}
```

Figure 45: C function

```
     r[4] = L[r[31] + a.];      /* load arg a into reg */
     r[5] = 0;                  /* j = 0; */
     r[3] = 0;                  /* k = 0; */
     r[2] = 0;                  /* i = 0; */
L16: cc = r[4] ? r[2];          /* compare a to i */
     PC = cc >= 0, L14;         /* if was >= then goto L14 */
     r[2] = r[2] + 1;           /* i++ (delay slot filled) */
     r[5] = r[5] + 1;           /* j++ */
L14: cc = r[2] ? 5;             /* compare i to 5 */
     PC = cc < 0, L16;          /* if was < then goto L16 */
     cc = r[5] ? 5;             /* compare j to 5 */
     PC = cc != 0, L19;         /* if was != then goto L19 */
     NL = NL;                   /* noop (delay slot not filled) */
     r[2] = 0;                  /* i = 0; */
L22: cc = r[4] ? -5;            /* compare a to -5 */
     PC = cc >= 0, L23;         /* if was >= then goto L23 */
     r[2] = r[2] + 1;           /* i++ (delay slot filled) */
     PC = L20;                  /* goto L20 */
     r[3] = r[3] + 6;           /* k += 6; (delay slot filled) */
L23: r[3] = r[3] + 7;           /* k += 7; */
L20: cc = r[2] ? 10;            /* compare i to 10 */
     PC = cc < 0, L22;          /* if was < then goto L22 */
     NL = NL;                   /* noop (delay slot not filled) */
L19: PC = RT;                   /* return to caller */
     r[0] = r[3];               /* set return value to k (delay slot filled)*/
```

Figure 46: RTLs for C Function with Delayed Branches

```
     b[4] = b[7];                           /* save return address */
     b[2] = b[0] + (L16 - LC5);             /* calc address of L16 */
LC5: b[1] = b[0] + (L14 - LC2);             /* calc address of L14 */
LC2: r[4] = L[r[31] + a.];                  /* load arg a into reg */
     r[5] = 0;                              /* j = 0; */
     r[3] = 0;                              /* k = 0; */
     r[2] = 0;                              /* i = 0; */
L16: b[7] = r[4] >= r[2], b[1] | b[0];      /* if a >= i then b[7] = L14 */
     r[2] = r[2] + 1;b[0] = b[7];           /* i++; PC = b[7] */
     r[5] = r[5] + 1;                       /* j++ */
L14: b[7] = r[2] < 5, b[2] | b[0];          /* if i < 5 then b[7] = L16 */
     b[7] = b[0] + (L19 - LC1);b[0] = b[7]; /* calc address of L19; PC = b[7] */
LC1: b[7] = r[5] != 5, b[7] | b[0];         /* if j != 5 then b[7] = L19 */
     b[1] = b[0] + (L23 - LC0);b[0] = b[7]; /* calc address of L23; PC = b[7] */
LC0: b[3] = b[0] + (L20 - LC4);             /* calc address of L20 */
LC4: b[2] = b[0] + (L22 - LC3);             /* calc address of L22 */
LC3: r[2] = 0;                              /* i = 0; */
L22: b[7] = r[4] >= -5, b[1] | b[0];        /* if j != 5 then b[7] = L23 */
     r[2] = r[2] + 1;b[0] = b[7];           /* i++; PC = b[7] */
     r[3] = r[3] + 6;b[0] = b[3];           /* k += 6; PC = L20 */
L23: r[3] = r[3] + 7;                       /* k += 7; */
L20: b[7] = r[2] < 10, b[2] | b[0];         /* if i < 10 then b[7] = L22 */
     NL = NL;b[0] = b[7];                   /* noop; PC = b[7] */
L19: r[0] = r[3];b[0] = b[4];               /* return k; */
```

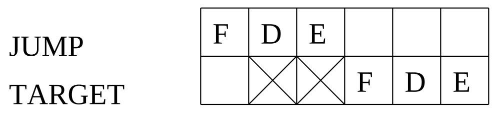
Figure 47: RTLs for C Function with Branch Registers

4.4.5. Reduction of Pipeline Delays

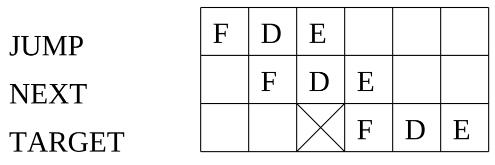
Most pipeline delays due to branches on conventional RISC machines can be avoided on this machine since branch target instructions are prefetched. Figure 48 contrasts the pipeline delays with a three stage pipeline for unconditional transfers of control on a pipelined machine without a delayed branch, with a delayed branch, and with branch registers. The three stages in the pipeline in this figure are:

1. Fetch
2. Decode
3. Execute

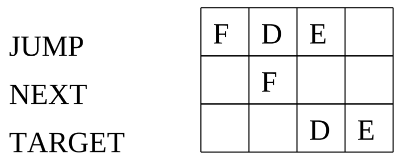
The branch target instruction cannot be fetched until its address has been calculated. For the first two machines, this occurs in the execute stage of the jump instruction. A conventional RISC machine without a delayed branch would have an *N-1* delay in the pipeline for unconditional transfers of control where *N* is the number of stages in the pipeline. The next instruction for the machine with a delayed branch and



(a) no delayed branch



(b) with delayed branch



(c) with branch registers

Figure 48: Pipeline Delays for Unconditional Transfers of Control

the machine with branch registers represents the next sequential instruction following the jump instruction. Thus, a RISC machine with a delayed branch, where the branch is delayed for one instruction, would have an *N-2* delay in the pipeline. Finding more than one useful instruction to place behind a delayed branch is difficult for most types of programs [McH86].

A jump instruction for the machine with branch registers is an instruction that references a branch register that is not the PC (`b[0]`). The branch register referenced is used during the decode stage of the jump instruction to determine which one of the set of instruction registers is to be input as the next instruction to be decoded. While the jump instruction is being decoded, the next sequential instruction is being fetched and loaded into `i[0]`, the default instruction register. If `b[0]` had been referenced, then `i[0]` would be input to the decode stage. Since a different branch register is referenced for the jump instruction, its corresponding instruction register containing the branch target instruction would be input to the next decode stage. Thus, assuming that the branch target instruction has been prefetched and is available in the appropriate instruction register, the machine with branch registers would have no pipeline delay for unconditional transfers of control regardless of the number of stages in the pipeline.

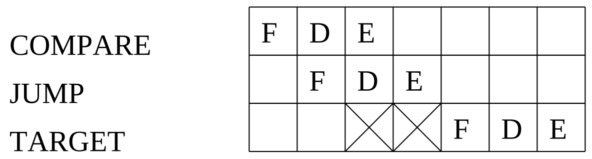
The example in Figure 49 shows the actions taken by each stage in the pipeline stage for an unconditional transfer of control in the branch register machine, assuming that the jump sequentially follows the previously executed instruction. The subscript on the actions denotes the stage of the pipeline. In the first stage, the jump instruction is fetched from memory and the PC is incremented to the next sequential instruction. In the second stage, the jump instruction is decoded and the next sequential instruction after the jump is fetched from memory. In the third stage, the jump instruction is executed, the prefetched instruction at the branch target in `i[4]` is decoded, and the instruction sequentially following the branch target is fetched. Since the address in a branch register is incremented after being used to prefetch an instruction from the cache, the branch register contains the address of the instruction after the branch target.

```
r[1] = r[1] + 1; b[0] = b[4];
```

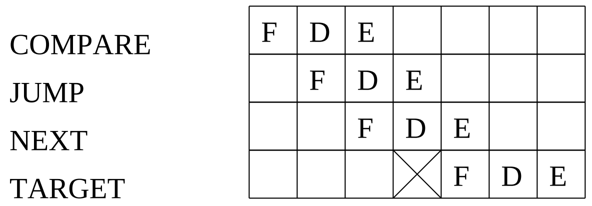
pipeline stage actions	JUMP	NEXT	TARGET	AFTER
(i[0] = M[b[0]]; b[0] = b[0] + 4;)$_F$	F			
(DECODE = i[0];)$_D$ (i[0] = M[b[0]]; b[0] = b[0] + 4;)$_F$	D	F		
(r[1] = r[1] + 1;)$_E$ (DECODE = i[4];)$_D$ (i[0] = M[b[4]]; b[0] = b[4] + 4;)$_F$	E		D	F

Figure 49: Pipeline Actions for Unconditional Transfer of Control

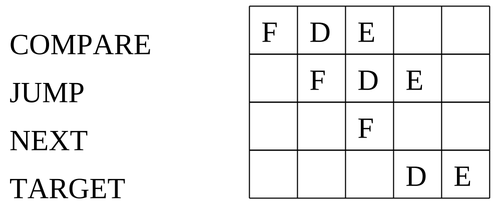
Figure 50 contrasts the pipeline delays for conditional transfers of control for the same three types of machines. As for unconditional transfers of control, the conventional RISC machine without a delayed branch would have a *N-1* pipeline delay and the RISC machine with a delayed branch would have a *N-2* pipeline delay for conditional transfers of control. The compare instruction for the machine with branch registers will assign one of two branch registers to a destination branch register depending upon the result of the condition in the compare. It will also make an assignment between the corresponding instruction



(a) no delayed branch



(b) with delayed branch



(c) with branch registers

Figure 50: Pipeline Delays for Conditional Transfers of Control

registers. The conditional jump instruction represents the instruction following the compare instruction that references the destination branch register of the compare instruction. The branch register referenced is used during the decode stage of the conditional jump instruction to cause the corresponding instruction register to be input as the next instruction to be decoded. Therefore, the decode stage of the target instruction cannot be accomplished until the last stage of the compare instruction is finished. This results in an *N-3* pipeline delay for conditional transfers of control for a machine with branch registers.

The example in Figure 51 shows the actions taken by each stage in the pipeline for a conditional transfer of control, assuming that the compare instruction sequentially follows the previously executed instruction. In the first stage, the compare instruction is fetched from memory and the PC is incremented to the next sequential instruction. In the second stage, the compare instruction is decoded and the jump instruction is fetched from memory. In the third stage, the compare instruction is executed (resulting in assignments to both `b[7]` and `i[7]`), the jump instruction is decoded, and the instruction sequentially following the jump is fetched. If the condition of the compare is not true, then `b[7]` and `i[7]` receive the same values from the fetch operation. In the fourth stage, either the target instruction or the next instruction after the jump is decoded and the instruction after the instruction being decoded is fetched.

To avoid pipeline delays, even when the branch target instruction is in the cache, the branch target address must be calculated early enough to be prefetched from the cache and placed in the instruction register before the target instruction is to be input to the decode stage. Assuming there is a one cycle delay between the point that the address is sent to the cache at the end of the execute stage and the instruction is loaded into the instruction register, this would require that the branch target address be calculated at least two instructions previous to the instruction with the transfer of control when the number of stages in the pipeline is three. This is shown in Figure 52.

4.4.6. Experimental Evaluation

In an attempt to reduce the number of operand memory references, many RISC machines have thirty-two or more general-purpose registers (e.g. MIPS-X, ARM, Spectrum). Without special compiler optimizations, such as inlining [Sch77] or interprocedural register allocation [Wal86], it is infrequent that

```
b[7] = r[5] < 0 -> b[3] | b[0];
r[1] = r[1] + 1; b[0] = b[7];
```

pipeline stage actions	COMPARE	JUMP	NEXT	TARGET	AFTER
(i[0] = M[b[0]]; b[0] = b[0] + 4;)$_F$	F				
(DECODE = i[0];)$_D$ (i[0] = M[b[0]]; b[0] = b[0] + 4;)$_F$	D	F			
(b[7] = r[5] < 0 -> b[3] | b[0] + 4; i[7] = r[5] < 0 -> i[3] | M[b[0]];)$_E$ (DECODE = i[0];)$_D$ (i[0] = M[b[0]]; b[0] = b[0] + 4;)$_F$	E	D	F		
(r[1] = r[1] + 1;)$_E$ (DECODE = i[7];)$_D$ (i[0] = M[b[7]]; b[0] = b[7] + 4;)$_F$		E		D	F

Figure 51: Pipeline Actions for Conditional Transfer of Control

ADDR CALC	F	D	E	╳		
INST		F	D	E		
JUMP			F	D	E	
NEXT				F		
TARGET					D	E

Figure 52: Prefetching to Avoid Pipeline Delays

a compiler can make effective use of even a majority of these registers for a function. In a previous study we calculated the number of data memory references that have the potential for being removed by using registers. We found that 98.5% could be removed by using only sixteen data registers. In order to evaluate this scheme, two machines were designed and then emulated using *ease*. The same set of test programs shown previously were used for this experiment.

The first machine used in the evaluation of the branch register approach served as a baseline to measure the second machine. The baseline machine was designed to have a simple RISC-like architecture. Features of this machine include:

1. 32-bit fixed-length instructions
2. reference to memory only by load and store instructions
3. delayed branches
4. 32 general-purpose data registers
5. 32 floating-point registers
6. three-address instructions
7. simple addressing modes

Figure 53 shows the instruction formats used in the baseline machine.

The second machine was a modification of the first to handle branches by using branch registers. Features of the branch register machine that differ from the baseline machine include:

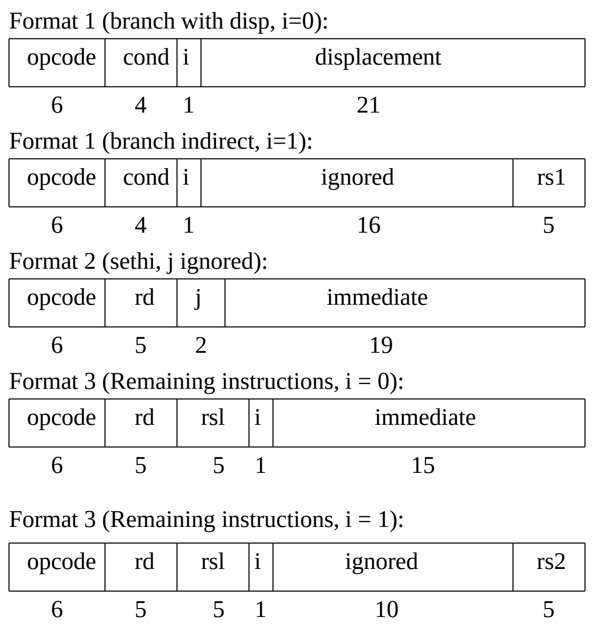

Format 1 (branch with disp, i=0):
opcode
cond
i
displacement
6
4
1
21
Format 1 (branch indirect, i=1):
opcode
cond
i
ignored
rs1
6
4
1
16
5
Format 2 (sethi, j ignored):
opcode
rd
j
immediate
6
5
2
19
Format 3 (Remaining instructions, i = 0):
opcode
rd
rsl
i
immediate
6
5
5
1
15
Format 3 (Remaining instructions, i = 1):
opcode
rd
rsl
i
ignored
rs2
6
5
5
1
10
5

Figure 53: Instruction Formats for the Baseline Machine

1. only 16 general-purpose data registers
2. only 16 floating-point registers
3. 8 branch registers
4. 8 instruction registers
5. no branch instructions
6. a compare instruction with an assignment
7. an instruction to calculate branch target addresses
8. smaller range of available constants in some instructions

If one ignores floating-point registers, there are approximately the same number of registers on each machine. Figure 54 shows the instruction formats used in the branch register machine. Since the only differences between the baseline machine and the branch register machine are the instructions to use branch registers as opposed to branches, the fewer number of data registers that can be referenced, and the smaller range of constants available, the reports generated by *ease* can accurately show the impact of using registers for branches.

The branch register machine executed 6.8% fewer instructions and yet performed 2.0% additional data memory references as compared to the baseline machine. The ratio of fewer instructions executed to

Format 1 (cmp with immed, i = 0):

opcode	cond	bs1	rs1	i	immediate	br
6	4	3	4	1	11	3

Format 1 (cmp with reg, i = 1):

opcode	cond	bs1	rs1	i	ignored	rs2	br
6	4	3	4	1	7	4	3

Format 2 (sethi, inst addr calc):

opcode	rd	immediate	br
6	4	19	3

Format 3 (Remaining instructions, i = 0):

opcode	rd	rsl	i	immediate	br
6	4	4	1	14	3

Format 3 (Remaining instructions, i = 1):

opcode	rd	rsl	i	ignored	rs2	br
6	4	4	1	10	4	3

Figure 54: Instruction Formats for the Branch Register Machine

additional data references for the branch register machine was 10 to 1. Approximately 14% of the instructions executed on the baseline machine were transfers of control. The reduction in the number of instructions executed was mostly due to moving branch target address calculations out of loops. Another factor was replacing 36% (almost 2.6 million) of the noops in delay slots of branches in the baseline machine with branch target address calculations at points of transfers of control in the branch register machine. There were also additional instructions executed on the branch register machine to save and restore branch registers. The additional data references on the branch register machine were due to both fewer variables being allocated to registers and saves and restores of branch registers. Table 28 shows the results from running the test set through both machines.

machine	millions of instructions executed	millions of data references
baseline	183.04	61.99
branch register	170.75	63.22
difference	-12.29	+1.23

Table 28: Dynamic Measurements from the Two Machines

By prefetching branch target instructions at the point the branch target address is calculated, delays in the pipeline can be decreased. In the baseline machine, there were 7.95 million unconditional transfers of control and 17.69 million conditional transfers of control. Assuming a pipeline of three stages, not uncommon for RISC machines [GiM87], then each branch on the baseline machine would require at least a one-stage delay. Also assuming that each instruction can execute in one machine cycle, and no other pipeline delays except for transfers of control, then the test set would require about 208.83 million cycles to be executed on the baseline machine. As shown previously in Figures 48 and 50, the branch register machine would require no delay for both unconditional and conditional branches in a three stage pipeline assuming that the branch target instruction has been prefetched. As shown in Figure 52, the branch target address must be calculated at least two instructions before a transfer of control to avoid pipeline delays even with a cache hit. We estimate that only 13.86% of the transfers of control that were executed would result in a pipeline delay. Thus, the branch register machine would require about 22.09 million (10.6%) fewer cycles to be executed due to fewer delays in the pipeline alone. There would be greater savings for machines having pipelines with more stages. For instance, we estimate that the branch register machine would require about 30.04 million (12.8%) fewer cycles to be executed due to fewer delays in the pipeline alone assuming a pipeline with four stages.

4.4.7. Hardware Considerations

An instruction cache typically reduces the number of memory references by exploiting the principles of spatial and temporal locality. However, when a particular main memory line is referenced for the first time, the instructions in that line must be brought into the cache and these misses will cause delays. When an assignment is made to a branch register, the value being assigned is the address of an instruction that will likely be brought eventually into the instruction cache.

To take advantage of this knowledge, each assignment to a branch register has the side effect of specifying to the instruction cache to prefetch the line associated with the instruction address. Prefetch requests could be performed efficiently with an instruction cache that would allow reading a line from main memory at the same time as requests for instruction words from the CPU that are cache hits are honored. This could be accomplished by setting a busy bit in the line of the cache that is being read from memory at the beginning of a prefetch request and setting it to not busy after the prefetch has completed. To handle prefetch requests would require a queuing mechanism with the size of the queue equal to the number of available branch registers. A queue would allow the cache to give priority to cache misses for sequential fetches over prefetch requests that do not have to wait. Directing the instruction cache to bring in instructions before they are used will not decrease the number of cache misses. It will, however, decrease or eliminate the delay of loading the instruction into the cache when it is needed to be fetched and executed.

The machine must determine if an instruction has been brought into an instruction register and thus is ready to be decoded after the corresponding branch register is referenced in the preceding instruction. This can be accomplished by using a flag register that contains a set of bits that correspond to the set of instruction registers. The appropriate bit could be cleared when the request is sent to the cache and set when the instruction is fetched from the cache. Note that this would require the compiler to ensure that branch target addresses are always calculated before the branch register is referenced.

4.4.8. Future Work

There are many interesting areas involving branch registers that remain to be explored. The best cache organization to be used with branch registers needs to be investigated. An associativity of at least two would ensure that a branch target could be prefetched without displacing the current instructions that are being executed. A larger number of words in a cache line may be appropriate in order to less often have cache misses of sequential instructions while instructions at a branch target are being loaded from memory into the instruction cache. Another feature of the cache organization to investigate is the total number of words in the cache. Since instructions to calculate branch target addresses can be moved out of loops, the number of instructions in loops will be fewer. This may improve cache performance in machines with small on-chip caches.

The exact placement of the branch target address calculation can affect performance. The beginning of the function could be aligned on a cache line boundary and the compiler would have information about the structure of the cache. This information would include

1. the cache line size
2. the number of cache lines in each set
3. the number of cache sets in the cache

Using this information the compiler could attempt to place the calculation where there would be less potential conflict between cache misses for sequential instructions and cache misses for prefetched branch targets. By attempting to place these calculations at the beginning of a cache line, the potential for conflict would be reduced.

Prefetching branch targets may result in some instructions being brought into the cache that are not used (cache pollution). Since most branches tend to be taken [LeS84], we have assumed that this penalty would not be significant. By estimating the number of cycles required to execute programs (which includes cache delays) on the branch register machine and the baseline machine, the performance penalty due to cache pollution of unused prefetched branch targets could be determined.

Other code generation strategies can be investigated. For instance, if a fast compare instruction could be used to test the condition during the decode stage [McH86], then the compare instruction could update the program counter directly. A bit may be used in the compare instruction to indicate whether to squash [McH86] the following instruction depending upon the result of the comparison. Eight branch registers and eight instruction registers were used in the experiment. The available number of these registers and the corresponding changes in the instruction formats could be varied to determine the most cost effective combination.

4.4.9. Conclusions

Using branch registers to accomplish transfers of control has been shown to be potentially effective. By moving the calculation of branch target addresses out of loops, the cost of performing branches inside of loops can disappear and result in fewer executed instructions. By prefetching the branch target instruction when the branch target address is calculated, branch target instructions can be inserted into the pipeline with fewer delays. By moving the assignment of branch registers away from the use of the branch register, delays due to cache misses of branch targets may be decreased. The performance of a small instruction cache, such as the cache for the CRISP architecture [DiM87b], could also be enhanced since the number of instructions in loops will be fewer. Enhancing the effectiveness of the code can be accomplished with conventional optimizations of common subexpression elimination and code motion. A machine with branch registers should also be inexpensive to construct since the hardware would be comparable to a conventional RISC machine.

CHAPTER 5

CONCLUSIONS

The first goal for this research was to provide an environment in which program measurements can be obtained without the problems of past architectural studies. This goal has been achieved by modifying *vpo* (Very Portable Optimizer) to capture instruction characteristics during compilation and to insert code to capture frequency counts during execution. Since instructions to increment counters were only inserted once for each set of basic blocks that are executed the same number of times, this method executes faster than any other known method that collects similar information. Because the information about instructions is collected as a side-effect of the compiler parsing instructions, the method only requires 10 to 20 percent overhead in compilation time. Since most of the modifications to *vpo* to collect measurements has been accomplished in a machine-independent fashion, the machine-dependent modifications can be ported to a new machine in a few hours. Detailed reports can be quickly produced for a variety of static and dynamic measurements. Because the method collects measurements on RTLs, rather than assembly or machine instructions, measurements can be collected on new instructions or many other architectural features that do not yet exist as long as they can be modeled on existing features.

The second goal was to use the environment to evaluate several architectures after establishing this environment on several current machines. *Ease* was ported to the following architectures:

1. DEC VAX-11
2. Harris HCX-9
3. AT&T 3B15
4. Motorola 68020/68881
5. National Semiconductor 32016
6. Intel 80386/80387
7. Concurrent 3230
8. IBM RT
9. Intergraph Clipper
10. SUN SPARC

Measurements were obtained and analyzed for each of the above architectures. No other architectural study had the combination of 1) using a large set of test programs, 2) examining several current architectures, 3) eliminating most compiler bias, 4) extracting and analyzing detailed measurements, and 5) comparing entire architectures by using generic costs.

It was found there were strong linear relationships between most of the static and dynamic measurements. Each of these relationships was examined and observations were discussed about why they occurred. Typically static measurements are easier to obtain and dynamic measurements give more useful information on performance. Therefore, regression analysis was used to produce an equation that estimates each dynamic measurement from its corresponding static measurement.

The last goal was to demonstrate the effectiveness of the environment by performing code generation and machine design experiments. Six different methods for saving and restoring registers between function activations were examined for the VAX-11 and 68020. The number of scratch and non-scratch registers for each method in each architecture was varied. It was found for those methods that either combined callee and caller-save approaches or performed more caller-flow analysis, additional scratch registers resulted in more effective code. Methods that used callee-flow analysis within the scope of a function to reduce the numbers of saves and restores had little impact. Since there are instances in a function when both a callee-save or a caller-save may be most effective, hybrid methods that combine both approaches produced better code.

The VAX-11 compiler was modified to pass up to six arguments through the scratch registers. It was found that all of the scratch registers could be used to pass arguments without displacing the use of these registers for other purposes. Passing arguments through registers as a calling sequence convention resulted in many more arguments being allocated to registers. Since most arguments were allocated to registers and at least two memory references were eliminated for each argument, the total number of memory references were reduced.

The VAX-11 compiler was also modified to use more primitive call and return instructions in addition to passing arguments through registers. It was discovered that many of the functions performed by the complex call and return instructions on the VAX-11 were unnecessary a large percentage of the time. Since arguments were passed through the scratch registers, the stack pointer was rarely adjusted after a call. This allowed optimizations to be applied when the primitive call instruction was followed by an unconditional jump, a sequence of instructions to return to the caller, or another call. By moving instructions when possible to proceed a call, these optimizations were available on 17.3% of the calls. The combination of passing arguments through registers, using primitive call and return instructions, and applying the new optimizations available resulted in over one fourth of the memory references from the execution of a set of test programs being eliminated.

To demonstrate that *ease* could be used for machine design, a new technique for reducing the cost of branches by using registers was investigated. In an attempt to reduce the number of operand memory references, many RISC machines have thirty-two or more general-purpose registers. Without special compiler optimizations, such as inlining or interprocedural register allocation, it is infrequent that a compiler can make effective use of even a majority of these registers for a function. To demonstrate the effectiveness of the branch register technique, two machines were designed and emulated using *ease*. The first machine was designed as a conventional simple RISC machine with delayed branches and thirty-two general-purpose registers used for data references. The second machine used the branch register technique for transfers of control and had sixteen data registers and sixteen registers used for branching. The measurements collected from the two machines show that the branch register technique has the potential for effectively reducing the cost of transfers of control.

The results of this dissertation should be useful in several areas. The method used to collect measurements executes efficiently and requires only simple modifications to the machine-dependent portions of a *vpo* optimizer. Thus, this method can be used to collect measurements by different compilers for future experiments or analysis. The raw measurements from the architectural study can be used by researchers or designers to draw their own conclusions on the effectiveness of architectural features. The differences observed between static and dynamic measurements may be used to estimate one type of measurement if the other type is unavailable. The insights gained from the analysis and experiments may be useful to architects designing new machines or modifying existing ones. They may also be useful to compiler writers to aid in designing a calling sequence or determining what types of optimizations may be most effective.

References

[AdZ89] T. Adams and R. Zimmerman, ‘‘An Anaylsis of 8086 Instruction Set Usage in MS-DOS Programs’’, *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, Ma, April 1989, 152-160.

[ASU86] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[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.

[BeN71] C. G. Bell and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, 1971.

[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.

[BLG83] M. L. Berenson, D. M. Levine and M. Goldstein, *Intermediate Statistical Methods and Applications*, Prentice-Hall, Inc, 1983.

[CAC81] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins and P. W. Markstein, ‘‘Register Allocation via Coloring’’, *Computer Languages 6*, 1 (1981), 47-57.

[ChH84] F. Chow and J. Hennessy, ‘‘Register Allocation by Priority-based Coloring’’, *Proceedings of the SIGPLAN Notices ’84 Symposium on Compiler Construction*, Montreal, Canada, June 1984, 222-232.

[Cho88] F. Chow, ‘‘Minimizing Register Usage Penalty at Procedure Calls’’, *Proceedings of the SIGPLAN Notices ’88 Symposium on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988, 85-94.

[ClL82] D. W. Clark and H. M. Levy, ‘‘Measurement and Analysis of Instruction Use in the VAX-11/780’’, *Proceedings of the 9th Annual Symposium on Computer Architecture*, Austin, Texas, April 1982, 9-17.

[CNO87] 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’’, *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, October, 1987, 180-192.

[CoL82] R. Cook and I. Lee, ‘‘A Contextual Analysis of Pascal Programs’’, *Software—Practice & Experience 12*, 2 (February 1982), 195-203.

[CoD82] R. Cook and N. Donde, ‘‘An Experiment to Improve Operand Addressing’’, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, March 1982, 87-91.

[Coo89] R. Cook, ‘‘An Empirical Analysis of the Lilith Instructions Set’’, *IEEE Transactions on Computers 38*, 1 (January 1989), 156-158.

[DaW88] J. W. Davidson and D. B. Whalley, ‘‘Creating a Back End for VPCC’’, RM-88-01, University of Virginia, Charlottesville, VA, July 1988.

[DaW89] J. Davidson and D. Whalley, ‘‘Quick Compilers Using Peephole Optimizations’’, *Software—Practice & Experience 19*, 1 (January 1989), 195-203.

[DRW89] J. Davidson, J. Rabung and D. Whalley, ‘‘Relating Static and Dynamic Machine Code Measurements’’, *IEEE Transactions on Computers*, submitted June 1989.

[DaW90a] J. W. Davidson and D. B. Whalley, ‘‘Ease: An Environment for Architecture Study and Experimentation’’, *Proceedings SIGMETRICS ’90 Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, May 1990.

[DaW90b] J. W. Davidson and D. B. Whalley, ‘‘Reducing the Cost of Branches by Using Registers’’, *Proceedings of the 17th Annual Symposium on Computer Architecture*, Seattle, Washington, May 1990.

[DaW91] J. Davidson and D. Whalley, ‘‘Methods for Saving and Restoring Register Values across Function Calls’’, *Software—Practice & Experience 21*, 2 (February 1991), 149-165.

[DDD81] *VAX Architecture Handbook*, Digital Equipment Corporation, Maynard, MA, 1981.

[Dit80] D. R. Ditzel, ‘‘Program Measurements on a High-Level Language Computer’’, *Computer 13*, 8 (August 1980), 62-72.

[DiM87a] D. R. Ditzel and H. R. McLellan, ‘‘Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero’’, *Proceedings of the 14th Annual Symposium on Computer Architecture*, Pittsburgh, Pennsylvania, June 1987, 2-9.

[DiM87b] D. R. Ditzel and H. R. McLellan, ‘‘The Hardware Architecture of the CRISP Microprocessor’’, *Proceedings of the 14th Annual Symposium on Computer Architecture*, Pittsburgh, Pennsylvania, June 1987, 309-319.

[Er83] M. Er, ‘‘Optimizing Procedure Calls and Returns’’, *Software—Practice & Experience 13*, 10 (October 1983), 921-939.

[Flo63] R. W. Floyd, ‘‘Syntax Analysis and Operator Precedence’’, *Journal of the ACM 10*, 3 (March 1963), 316-333.

[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.

[GiM87] C. Gimarc and V. Milutinovic, ‘‘A Survey of RISC Processors and Computers of the Mid-1980s’’, *IEEE Computer 20*, 9 (September 1987), 59-69.

[GKM82] S. L. Graham, P. B. Kessler and M. K. McKusick, ‘‘gprof: A Call Graph Execution Profiler’’, *Proceedings SIGPLAN Notices ’82 Symposium on Compiler Construction*, Boston, MA, June 1982, 120-126.

[GrB82] M. L. Griss and E. Benson, ‘‘Current Status of a Portable Lisp Compiler’’, *Proceedings of the SIGPLAN Notices ’82 Symposium on Compiler Construction*, Boston, MA, June 1982, 276-283.

[Han85] D. R. Hanson, ‘‘Compact Recursive-descent Parsing of Expressions’’, *Software—Practice and Experience 15*, 12 (December 1985), 1205-1212.

[HeG83] J. Hennessy and T. Gross, ‘‘Postpass Code Optimization of Pipeline Constraints’’, *ACM Transactions on Programming Languages and Systems 5*, 3 (July 1983), 422-448.

[Hen84] J. L. Hennessy, ‘‘VLSI Processor Architecture’’, *IEEE Transactions on Computers 33*, 12 (December 1984), 1221-1246.

[Huc83] J. C. Huck, *Comparative Anaylsis of Computer Architectures*, PhD Dissertation, Stanford University, August 1983.

[HuL86] M. Huguet and T. Lang, ‘‘Reduced Register Saving/Restoring in Single-Window Register Files’’, *Computer Architecture News 14*, 3 (June 1986), 17-26.

[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 SIGPLAN Notices ’87 Symposium on Interpreters and Interpretive Techniques*, St. Paul, Minnesota, June 1987, 14-25.

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

[Kat83] M. G. H. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*, PhD Dissertation, University of California, Berkeley, CA, 1983.

[Knu71] D. E. Knuth, ‘‘An Empirical Study of FORTRAN Programs’’, *Software—Practice and Experience 1*, 2 (April 1971), 105-133.

[KKR86] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin and N. Adams, ‘‘ORBIT: An Optimizing Compiler for Scheme’’, *Sigplan Notices 21*, 7 (July 1986), 207-218.

[LeS84] J. K. F. Lee and A. J. Smith, ‘‘Branch Prediction Strategies and Branch Target Buffer Design’’, *IEEE Computer 17*, 1 (January 1984), 6-22.

[Lun77] A. Lunde, ‘‘Empirical Evaluation of Some Features of Instruction Set Processor Architectures’’, *Communications of the ACM 20*, 3 (March 1977), 143-153.

[McD82] G. McDaniel, ‘‘An Analysis of a Mesa Instruction Set Using Dynamic Instruction Frequencies’’, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, March 1982, 167-176.

[McH86] S. McFarling and J. Hennessy, ‘‘Reducing the Cost of Branches’’, *Proceedings of the 13th Annual Symposium on Computer Architecture*, Tokyo, Japan, June 1986, 396-403.

[Nor86] M. J. Norusis, *SPSS/PC+*, McGraw-Hill, New York, NY, 1986.

[PaS82] D. A. Patterson and C. H. Sequin, ‘‘A VLSI RISC’’, *IEEE Computer 15*, 9 (September 1982), 8-21.

[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.

[Pat85] D. A. Patterson, ‘‘Reduced Instruction Set Computers’’, *Communications of the ACM 28*, 1 (January 1985), 8-21.

[PeS77] B. L. Peuto and L. J. Shustek, ‘‘An Instruction Timing Model of CPU Performance’’, *Proceedings of the 4th Annual Symposium on Computer Architecture*, Silver Spring, Maryland, March 1977, 165-178.

[Pow84] M. L. Powell, ‘‘A Portable Optimizing Compiler for Modula-2’’, *Proceedings of the SIGPLAN Notices ’84 Symposium on Compiler Construction*, Montreal, Canada, June 1984, 310-318.

[RaR77] B. R. Rau and G. E. Rossman, ‘‘The Effect of Instruction Fetch Strategies upon the Performance of Pipelined Instruction Units’’, *Proceedings of the 4th Annual Symposium on Computer Architecture*, Silver Spring, Maryland, March 1977, 80-89.

[RiF72] E. M. Riseman and C. C. Foster, ‘‘The Inhibition of Potential Parallelism by Conditional Jumps’’, *IEEE Transactions on Computers 21*, 12 (December 1972), 1405-1411.

[Ros69] R. F. Rosin, ‘‘Contemporary Concepts of Microprogramming and Emulation’’, *Computing Surveys 1*, 4 (December 1969), 197-212.

[SSS87] *The SPARC Architecture Manual*, SUN Microsystems, Inc., Mountain View, CA, 1987.

[Sch77] R. W. Scheifler, ‘‘An Analysis of Inline Substitution for a Structured Programming Language’’, *Communications of the ACM 20*, 9 (September 1977), 647-654.

[SwS82] R. E. Sweet and J. G. Sandman, Jr., ‘‘Empirical Analysis of the Mesa Instruction Set’’, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, March 1982, 158-166.

[Tan78] A. S. Tanenbaum, ‘‘Implications of Structured Programming for Machine Architecture’’, *Communications of the ACM 21*, 3 (March 1978), 237-246.

[Wal86] D. W. Wall, ‘‘Global Register Allocation at Link Time’’, *Proceedings of the SIGPLAN Notices ’86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986, 264-275.

[Wat86] J. S. Watts, *Construction of a Retargetable C Language Front-end*, Masters Thesis, University of Virginia, Charlottesville, VA, 1986.

[Wei84] P. Weinberger, ‘‘Cheap Dynamic Instruction Counting’’, *AT&T Bell Laboratories Technical Journal 63*, 10 (October 1984), 1815-1826.

[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.

[WiS53] M. Wilkes and J. Stringer, ‘‘Microprogramming and the Design of the Control Circuits in an Electronic Digital Computer’’, *Proceedings of the Cambridge Philosophical Society*, Cambridge, England, April 1953.

[Win73] R. O. Winder, ‘‘A Data Base For Computer Performance Evaluation’’, *Computer 6*, 3 (March 1973), 25-29.

[Wir86] N. Wirth, ‘‘Microprocessor Architectures: A Comparison Based on Code Generation by Compiler’’, *Communications of the ACM 29*, 10 (October 1986), 978-990.

Abstract

This dissertation describes an environment for the evaluation of computer architectures and architectural features. Retargeting *ease* (Environment for Architecture Study and Experimentation) for a new machine requires little effort and permits detailed architectural-level measurements to be collected quickly. *Ease* uses a portable optimizer that performs transformations on a machine-independent representation of a program. Since code to perform instruction selection and most optimizations is constructed automatically, the quality of the code generated by the compiler has less dependence on the skill of the implementors than compilers constructed using other techniques. This allows different architectures to be compared with little compiler bias. The environment facilitates experimentation with new architectural features since the compilers using this system may be easily modified and instructions are represented in a hardware description language.

Ease was used to perform an architectural study of ten contemporary architectures. Three of the architectures are CISC machines, three are RISC machines, and the remaining four are of intermediate complexity. Both static and dynamic measurements were collected from a large set of test programs. Observations of the dynamic measurements collected from the architectures are discussed. By assigning generic costs to instructions and addressing modes that are similar across the different architectures, the costs of executing the test set for each architecture were compared.

Ease was used to perform several other studies that provided valuable information. These studies included finding linear relationships between specific static and dynamic machine code measurements and experiments to evaluate different code generation strategies. To demonstrate that *ease* could be used for machine design, a new technique for reducing the cost of branches by using registers was investigated.

Acknowledgements

I would like to thank several people who have given me support during my work for this degree. Special appreciation goes to my advisor, Dr. Jack Davidson, for his patience and guidance. His insight has not only given me a topic for a successful dissertation, but also an environment on which to base future research. I am sure the experience from working with him will aid me throughout my career.

I also would like to thank the other members of my committee for their advice and assistance. Dr. Cook’s suggestion to use a statistical approach for evaluating the measurements was very valuable. I thank Dr. Williams for his suggestions on the branch register experiment. I also appreciate the careful evaluation of my work by Dr. Grimshaw and Dr. O’Bagy.

Many students have also given me support in various ways. I especially thank Anne Holler for her advice and suggestions. She was always patient while listening to my problems or wild ideas. This research would not have been possible without the optimizer developed by Ricky Benitez. I also appreciate his quick assistance to any of the problems that were encountered. I do not have space to list the names of all of my other friends who have given me their encouragement during this process. I would like, however, to specifically thank Rob Renner and Chris Milner.

I also wish to express my gratitude to my family for their encouragement. More than anyone else, I thank my wife, Jiraporn, and daughter, Arada. I greatly appreciate the sacrifices they have made in order for me to come back to graduate school. The support and love they have given me have always made the difficult moments much easier to handle.

Table of Contents

List of Tables

List of Figures

Ease: An Environment for

Architecture Study and Experimentation

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

David B. Whalley

May 1991

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy (Computer Science)

David B. Whalley

This dissertation has been read and approved by the Examining Committee:

Dissertation Advisor

Committee Chairman

Accepted for the School of Engineering and Applied Science:

Dean, School of Engineering
and Applied Science

May 1991